MCR-81-515 NASA CR-181641

DIGITAL AVIONICS DESIGN AND RELIABILITY ANALYZER

NASA LaRC NAS1-15780

February 1981

Approved:

Edward C. Stanke, II
Program Manager

TABLE OF CONTENTS

		Page
1.0	Introduction	1-1
2.0	Applicable Documents	2-1
2.1	Reference Documents	2-1
2.2	Standards	2-1
2.3	Other	2-1
3.0	System Functional/Operational Description	3-1
3.1	Introduction	3-1
3.2	Usage Phases	3-2
3.3	Test Design Phase	3-3
3.4	Test Execution Phase	3-3
3.5	Data Reduction/Analysis Phase	3-13
4.0	System Specification	4-1
4.1	General System Configuration	4-1
4.2	Hardware Configuration	4-3
4.3	Software Configuration	4-11
Appendix A	Hardware Composition Trade Study	A-1
Appendix B	Microprogrammable Computer Trade Study	B-1
Attachment 1	Interim Technical Report	

List of Figures

Figure		Page
3-1	. . . . .	3-3
3-2	. . . . .	3-4
3-3	. . . . .	3-5
3-4	. . . . .	3-8
3-5	. . . . .	3-11
3-6	. . . . .	3-14
4-1	. . . . .	4-1
4-2	. . . . .	4-2

List of Tables

Table		Page
A-1	. . . . .	A-1
B-1	. . . . .	B-4
B-2	. . . . .	B-5

1.0 INTRODUCTION:

This document contains the description and specifications for a digital avionics design and reliability analyzer. It is the result of the study done by Martin Marietta concerning the use of emulation for investigating reliability and fault-tolerance issues for proposed highly reliable commercial digital avionics systems. The study was contracted by the NASA Langley Research Center because of the coming technology in commercial aircraft, which largely precludes traditional approaches to certification.

Airframes for the 1990's are designed to be much more fuel efficient than current designs, but this fuel efficiency is bought at a price of less stability. To maintain safe flight, very reliable avionics computers are envisioned to allow the necessary quick reaction times and continuous monitoring of flight parameters.

The computers are designed to break down so rarely (less than once in a human lifetime) that conventional bench and field tests cannot certify their reliability. The Federal Aviation Administration is in the process of adopting new certification procedures that emphasize mathematical models and simulations of the system over actual tests. To put the effort in perspective, the computers will be predicted to break down less often than the wings are expected to fall off planes in flight. The new avionics computers must be significantly more reliable than today's avionics computers. They could function unattended, despite hardware or software failures for at least a 10-hour flight. This super-reliability will be gained through redundant hardware and software. Faults that occurred will be counteracted automatically by hardware and/or software algorithms. As these highly fuel efficient aircraft would fly 100 percent of the time in critically stable conditions, control of the aircraft must be maintained concurrently with the fault detection and correction process. Further, any faults occurring during the recognition and correction of a previous fault must be handled as well.

The hardware/software configuration described in this document is referred to as the Digital Avionics Design and Reliability Analyzer. Its basic function is to provide for the simulation and emulation of the various fault-tolerant digital avionic computer designs that are developed. It has been established that hardware emulation at the gate-level will be utilized. The primary benefit of emulation to reliability analysis is the fact that it provides the capability to model a system at a very detailed level. This means that rather than basing reliability analyses on manufacturer's supplied data, or on expected probability distributions of failures of parts to determine the response of a system, detailed models of a system may now be employed on an experimental basis and system responses to faults observed rather than predicted. Emulation allows the direct insertion of faults into the system, rather than waiting for actual hardware failures to occur. This allows for controlled and accelerated testing of system reaction to hardware failures.

This report has two primary sections. Section 3 is a description of the functions of the system. This is intended to provide a perspective of the system for the specification which follows in Section 4. Section 4 contains the more definitive hardware and software requirements necessary to achieve the goals and functions given in Section 3.

There are two Appendices and one Attachment. Appendix A is the trade study which leads to the decision to specify a two machine system, including an emulation computer connected to a general purpose computer. Appendix B is an evaluation of potential computers to serve as the emulation computer. Attachment 1 is the previously delivered Interim Technical Report. This report details the feasibility study and describes in some detail the NASA Langley gate level algorithm which provided the basis for most of the performance figures required in the specification.

2.0 APPLICABLE DOCUMENTS

2.1 Reference Documents

1) Feasibility Study Report, Digital Avionics Design and Reliability Analyzer, November 1979.
2) Interim Technical Report, Digital Avionics Design and Reliability Analyzer, February 1980.
3) System Design Progress Report, Digital Avionics Design and Reliability Analyzer, July 1980.

2.2 Standards

1) Electronics Industries Association Standard RS-449
2) Federal Standard 1031
3) Electronics Industries Association Standard RS-232-C
4) American National Standards Institute X.3.9-1966
5) Federal Information Processing Standards Publication 1
6) Federal Information Processing Standards Publication 2
7) Federal Information Processing Standards Publication 3-1
8) Federal Information Processing Standards Publication 25
9) Federal Information Processing Standards Publication 16
10) Federal Information Processing Standards Publication 17
11) Federal Information Processing Standards Publication 18

2.3 Other

To be furnished by the Government

3.0 SYSTEM FUNCTIONAL/OPERATIONAL DESCRIPTION

3.1 Introduction

This section is intended to provide an overall description of what the system (including the analyst) must do without regard to the elements; hardware, software or manual procedures, which allow it to be done. The emphasis in this section is on the logical functions required for the digital avionics design and reliability analyzer. To express these functions, we use structured analysis tools and notation.[1] The notation which will be used throughout this section is based on three elements: data flow diagrams, mini-specifications, and the data dictionary.

3.1.1. Data Flow Diagrams

Data Flow Diagrams (DFD) are used to present the system pictorially thus reducing the amount of narrative needed. A DFD is a network representation of a system. The system may be automated, manual, or mixed. The DFD portrays the system in terms of its component functional pieces with all interfaces among the components indicated. A DFD does not represent the flow of control or the order of processing. Numbers used on the diagrams are for identification purposes only. Data Flow Diagrams are made up of four basic elements:

1) Data flows, represented by named vectors, are pipelines through which packets of information of known composition flow.

2) Processes, represented by bubbles, are transformations of incoming data flow(s) into outgoing data flow(s). Each process bubble needs a descriptive name.

3) Data stores, represented by two straight horizontal lines, are temporary repositories of data and may consist of tapes, discs, card sets, index files, data bases,or even someone's memory.

4) Data sources and sinks, represented by boxes, are persons, organizations, or other entities lying outside the context of a system, that are net originators or receivers of system data. A source box exists only to provide commentary about the system's connection to the outside world.

Data Flow Diagrams are expressed in levels. The first level, called the Context Diagram is labeled Diagram 0 and portrays an overall picture of the system with subsystems shown. These subsystems are labeled 1 through N. The subsystems are broken down in separate DFDs and further described. The components of the first subsystem are labeled 1.1, 1.2, 1.3, etc. When a subsystem has been decomposed to as simple a form as necessary. it is called a functional primitive.

1. Tom DeMarco, Structured Analysis and System Specification. New York; Yourdon, 1978.

There are many advantages to using leveled Data Flow Diagrams. They allow a top-down approach to analysis. By reading the top few levels one can get the big picture, or one can begin with the abstract and go to the detailed and narrow in on particular areas of interest. Each page is a complete presentation of the area of work allocated to it. All diagrams can be restricted to 8 1/2 X 11 inch paper.

3.1.2 Mini-Specification

The second part of the system functional definition consists of the Mini-Specifications which are concise descriptions of the bottom-level bubbles (functional primitives). Each Mini-Spec describes rules governing transformation of data flows arriving at the associated primitive into data flows leaving it.

3.1.3 Data Dictionary

To augment the Data Flow Diagram, there is an entity called the Data Dictionary. This contains rigorous definitions of all Data Flow Diagram elements such as data flows, components of data flows, files, and processes. These definitions relate all data elements through sequence, selection, or iteration.

3.1.4 The structured analysis information in this section is augmented as necessary by textual material to highlight important points.

3.2 Usage Phases

The digital avionics design and reliability analyzer is intended to support three primary uses:

1) Reliability analyses

2) Failure effects analyses

3) Conventional performance analyses

Regardless of their differences, each of these has several characteristics in common with the others. Primary among these commonalities is the fact that each involves data gathering which is facilitated by the technology of emulation. As shown in Figure 3-1, there are 3 basic phases of each use. These phases are:

1) Test design
2) Test execution
3) Data reduction/analysis

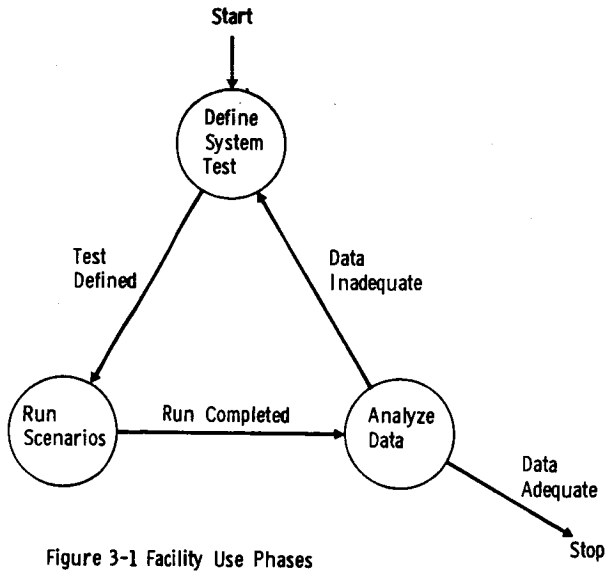


Figure 3-1 Facility Use Phases

These phases are shown in a different form in the Context Data Flow diagram given in Figure 3-2. In this diagram, the results of each phase are shown. Test design encompasses processes 1 and 3, test execution is process 2 and data reduction is process 4. Model building, process 1, is an inherent part of test design and so is not considered a separate phase in itself.

3.3. Test Design Phase

The modeling part of the Test Design Phase is shown graphically in Figure 3-3, and described in the process descriptions. One key concept which needs highlighting is the division of a system into functional blocks. This partitioning is necessary due to the time constraints of emulating at the gate level. Based on the results of the feasibility study (see Attachment 1), it is impossible to emulate the gate structure of the entire system under test. Thus the mixed mode concept, where the system is simulated at a functional level until a fault is inserted at which time the functional simulation of the affected block is replaced with a gate level emulation of that block.

Following Figure 3-3, Figure 3-4, and Figure 3-5 are mini-specifications describing each process shown in these figures.

One other concept not shown explicitly concerns the redundant computations which occur in a fault tolerant computing system. In a model, there is no necessity of actually performing redundant operations until one of the redundant paths errs (due to the introduction of a fault). This concept arises also during the Test Execution.

3.4 Test Execution Phase

The Test Execution Phase is shown in Figure 3-5. As noted in 3.3, the actual execution uses a combination of functional level simulation and gate level emulation of the machine under test.

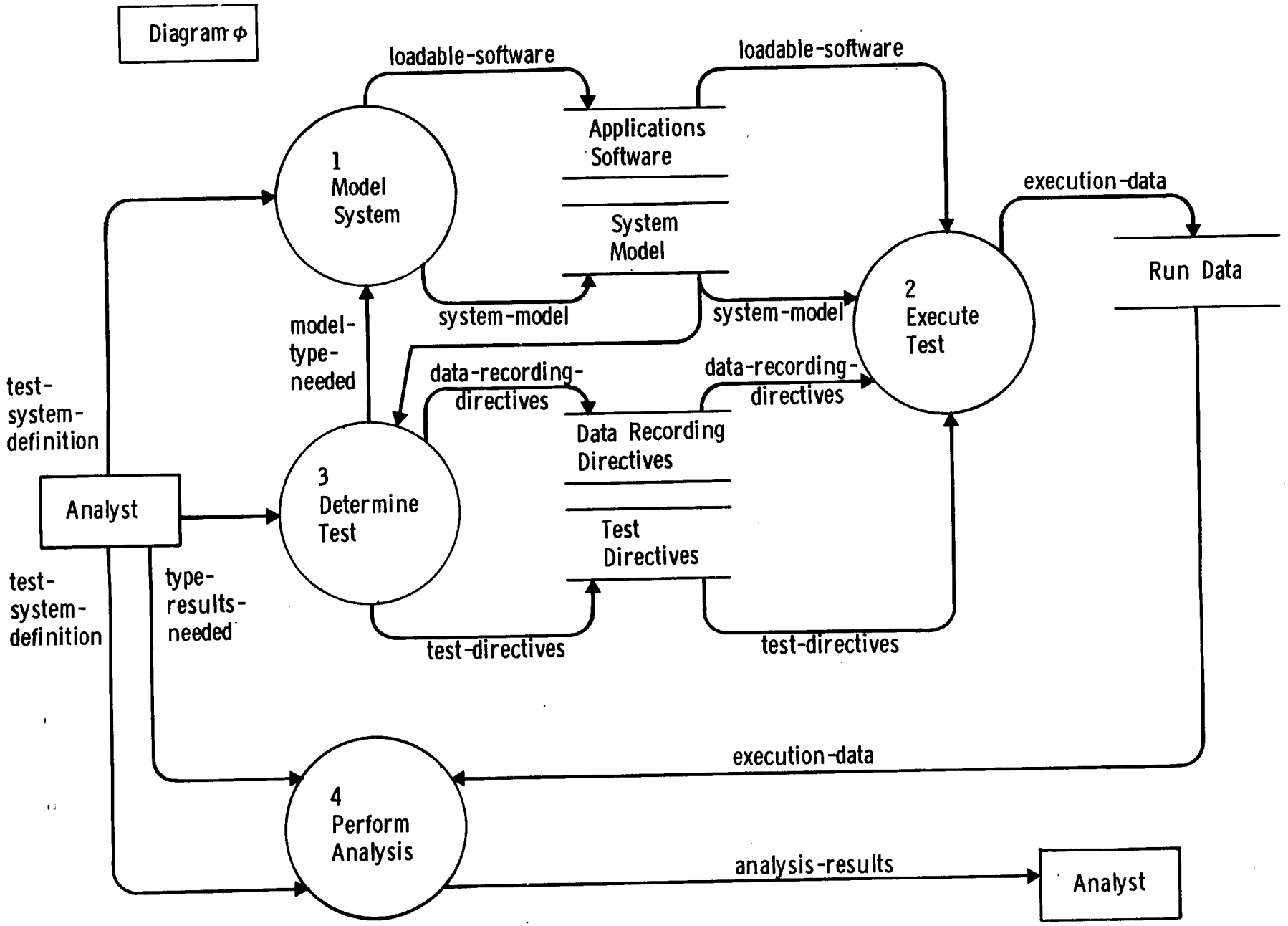


Figure 3.2 Perform Reliability And Design Analysis

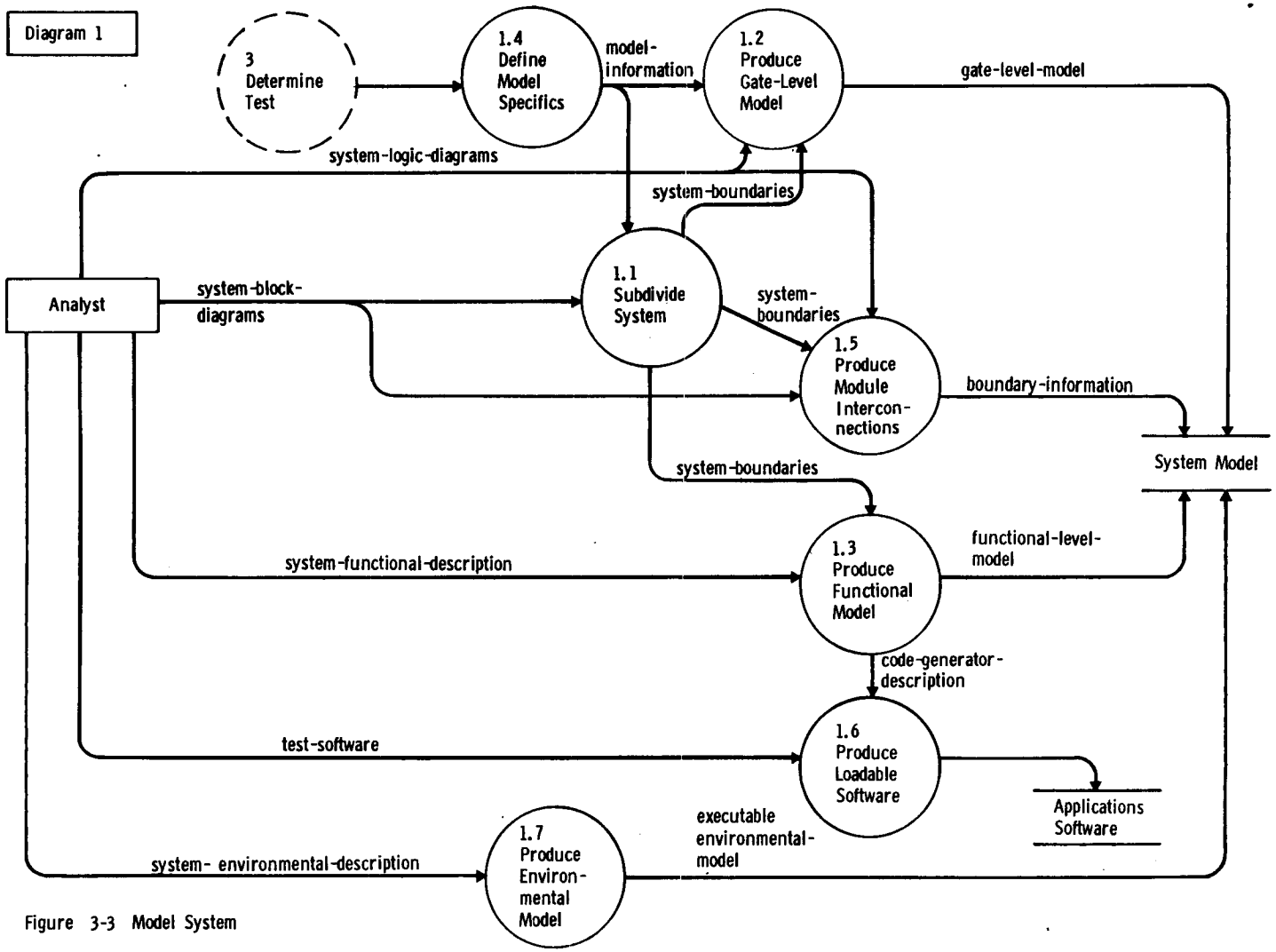


Figure 3-3 Model System

```
PROCESS: 1.1, Subdivide System

;PROCESS SPECIFICATION

IF model-information CONTAINS "gate-level-model-needed" THEN
   CREATE functional-block USING system-block-diagrams
   DEFINE system-boundry USING functional-block
   DEFINE internal-interfaces USING (functional-block AND
                                          system-block-diagrams)
ELSE
   DEFINE internal-interfaces USING system-block-diagrams
ENDIF
DEFINE external-interfaces USING system-block-diagrams

ENDPROCESS


PROCESS: 1.2, Produce Gate Level Model

;PROCESS SPECIFICATION

IF model-information CONTAINS "gate-level-model-needed" THEN
   FOR EACH functional-block IN system-boundries DO
      CREATE block-gate-model USING system-logic-diagrams
      TRANSLATE block-gate-model TO block-gate-table
   ENDFOR
   ASSEMBLE gate-level-model FROM block-gate-tables
ENDIF

ENDPROCESS


PROCESS: 1.3, Produce Functional Model

;PROCESS SPECIFICATION

FOR EACH functional-block IN system-boundries DO
   CREATE (block-functional-model AND interface-behavior-model) USING
                (system-functional-description AND internal-interfaces)
   TRANSLATE (block-functional-model AND interface-behavior-model) TO
          (functional-level-simulation-code AND functional-level-symbol-table)
ENDFOR
ASSEMBLE functional-level-model FROM (functional-level-simulation-code AND
                          functional-level-symbol-table)
CREATE code-generation-description USING system-functional-description

ENDPROCESS
```

```
PROCESS: 1.4, Define Model Specifics

;PROCESS SPECIFICATION

IF model-type-needed CONTAINS "gate-level-model-needed" THEN
   SET model-information TO "gate-level-model-needed" +
                                 "model-subdivision-needed"
ELSE
   SET model-information TO "monolithic-model-needed"
ENDIF

ENDPROCESS


PROCESS: 1.5, Produce Model Interconnection

;PROCESS SPECIFICATION

FOR EACH system-boundry DO
   DEFINE boundry-information USING (system-block-diagrams AND
                                     system-logic-diagrams)
ENDFOR

ENDPROCESS


PROCESS: 1.6, Produce Loadable Software

;PROCESS SPECIFICATION

FOR EACH test-software DO
   TRANSLATE test-software TO (machine-object-code AND symbol-table)
          USING code-generation-description
ENDFOR
ASSEMBLE loadable-software FROM (machine-object-code AND symbol-table)

ENDPROCESS


PROCESS: 1.7, Produce Environmental Model

;PROCESS SPECIFICATION

CREATE environmental-model-description USING
                          system-environmental-description
TRANSLATE environmental-model-description TO executable-environmental-model

ENDPROCESS
```

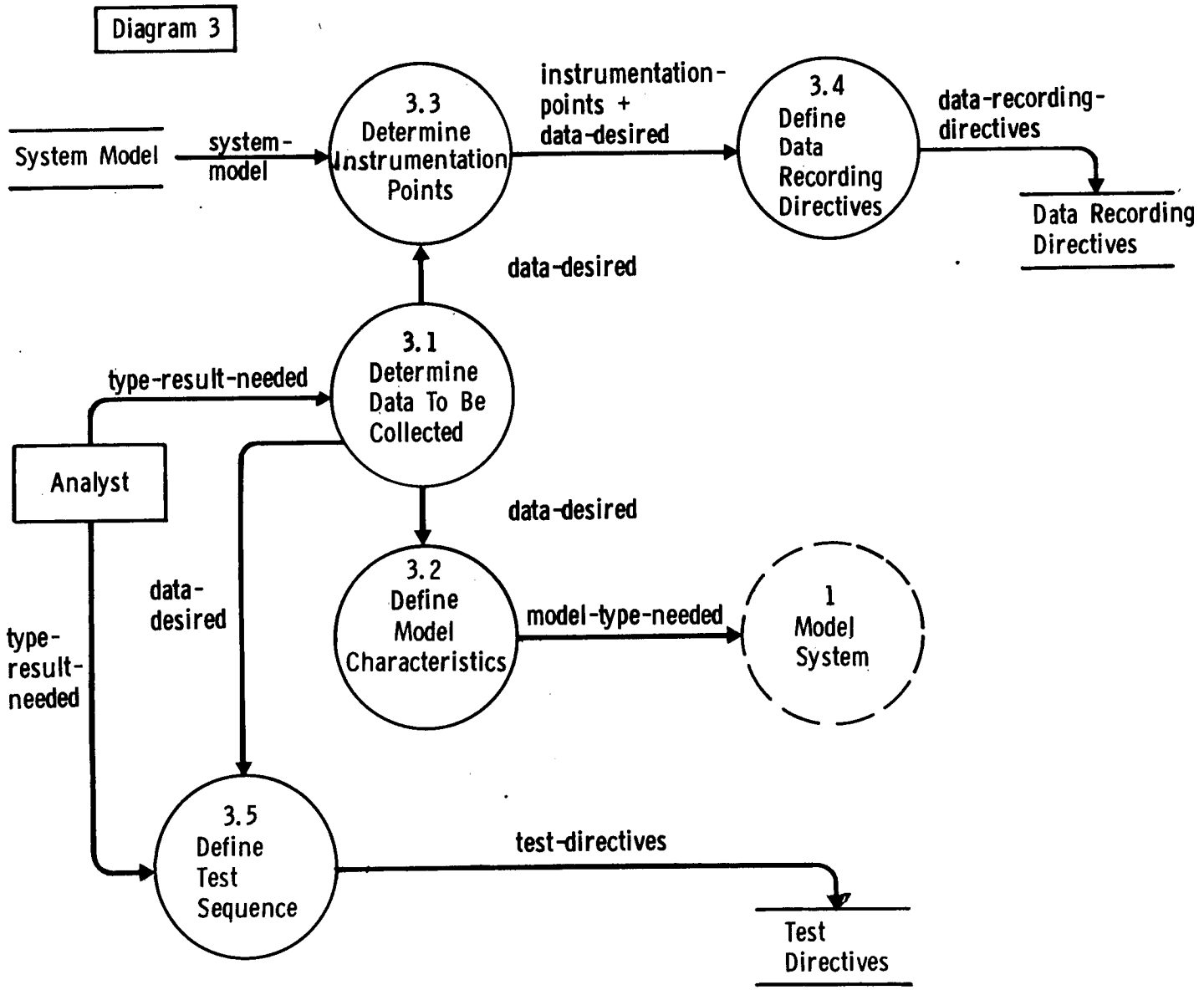


Figure 3-4 Determine Test

```
PROCESS: 3.1, Determine Data to be Collected

;PROCESS SPECIFICATION

IF type-result-needed = reliability-number THEN
   DETERMINE confidence-level-desired
   CALCULATE number-of-samples-necessary FROM confidence-level-desired
   DETERMINE type-data-necessary /* for statistical reduction */
   DETERMINE (type-of-failure-desired AND desired-failure-distribution)
ELSE
   IF type-result-needed = failure-effects-analysis THEN
      DETERMINE number-of-samples-necessary
      DETERMINE type-data-necessary FROM failure-mode-of-interest
      FOR EACH number-of-samples-necessary DO
         DETERMINE type-of-failure-desired
      ENDFOR
   ELSE
      IF type-result-needed = performance-characteristic THEN
         DETERMINE type-data-necessary /* for specific characteristics */
      ENDIF
   ENDIF
ENDIF

ENDPROCESS


PROCESS: 3.2, Define Model Characteristics

;PROCESS SPECIFICATION

IF type-data-necessary IN data-desired CONTAINS "gate-performance" THEN
   SET model-type-needed TO "functional-model-needed" +
                            "gate-level-model-needed"
ELSE
   SET model-type-needed TO "functional-model-needed"
ENDIF

ENDPROCESS


PROCESS: 3.3, Determine Instrumentation Points

;PROCESS SPECIFICATION

DETERMINE instrumentation-points IN (executable-environmental-model AND
                                     functional-level-model) USING
                                    type-data-necessary IN data-desired
IF type-data-necessary IN data-desired CONTAINS "gate-performance" THEN
   DETERMINE instrumentation-points IN gate-level-model
ENDIF

ENDPROCESS
```

```
PROCESS: 3.4, Define Data Recording Directives

;PROCESS SPECIFICATION

FOR EACH instrumentation-point DO
   DEFINE data-recording-directives USING data-desired
ENDFOR

ENDPROCESS


PROCESS: 3.5, Define Test Sequence

;PROCESS SPECIFICATION

IF type-result-needed = reliability-number THEN
   FOR EACH number-of-samples-necessary DO
      DETERMINE faults-to-be-inserted FROM (desired-failure-distribution AND
                                            specific-system-portion-of-interest)
   ENDFOR
ELSE
   IF type-result-needed = failure-effects-analysis THEN
      DETERMINE faults-to-be-inserted FROM (type-of-failure-desired AND
                                            specific-system-portion-of-interest)
   ENDIF
ENDIF
DETERMINE environmental-model-directives /* for desired test */

ENDPROCESS
```

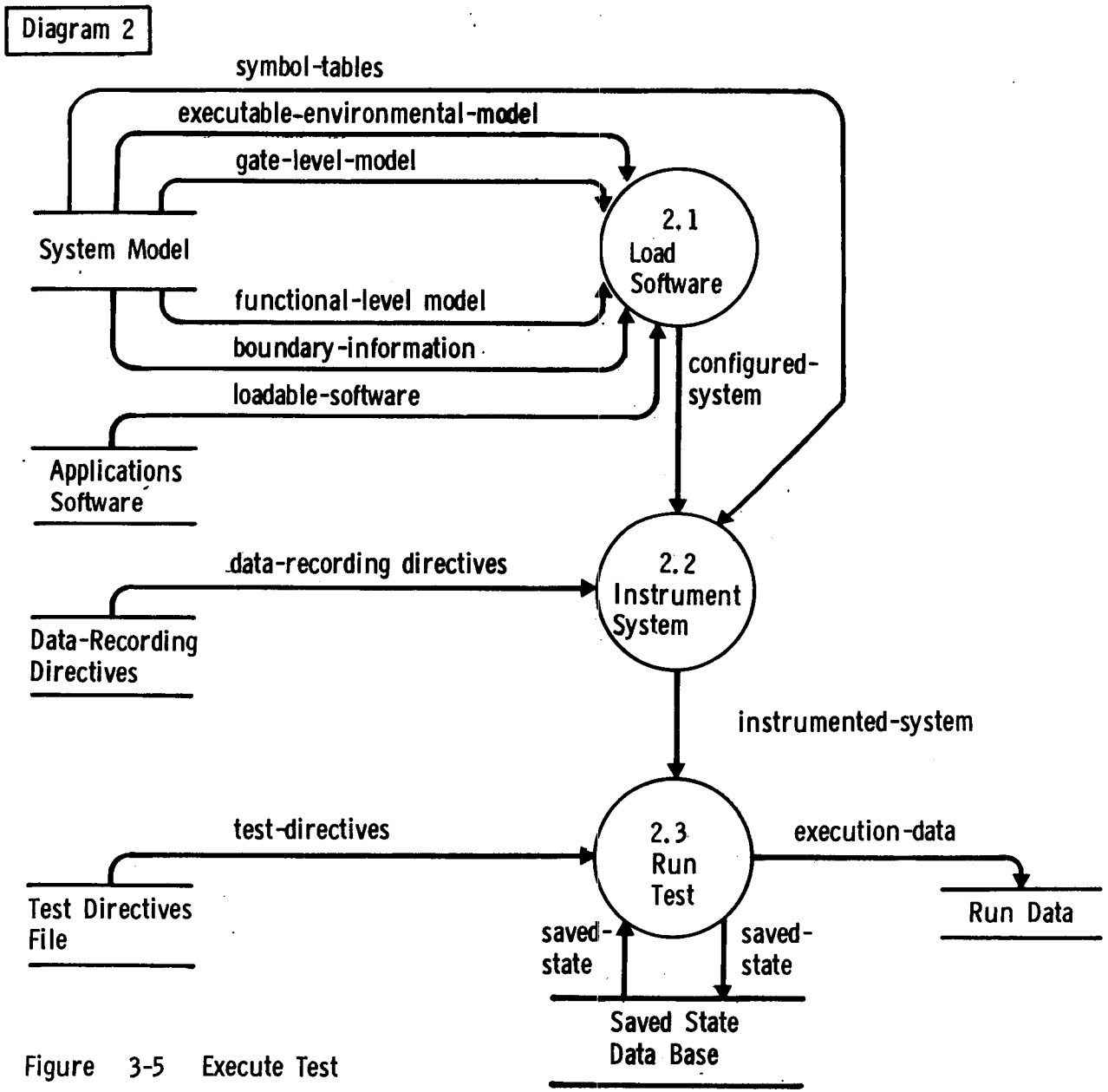


Figure 3-5 Execute Test

```
PROCESS: 2.1, Load Software

;PROCESS SPECIFICATION

ASSEMBLE configured-system FROM (executable-environmental-model +
                                 (functional-level-model + (gate-level-model +
                                 (boundry-information + loadable-software))))

ENDPROCESS


PROCESS: 2.2, Instrument System

;PROCESS SPECIFICATION

FOR EACH data-recording-directive DO
   IF data-recording-point = "symbol" THEN
      FIND insertion-point IN symbol-table
   ELSE
      IF data-recording-point = "target-memory-location" THEN
         FIND insertion-point USING functional-level-symbol-table
      ENDIF
   ENDIF
   CHANGE machine-object-code TO "trap"
   OUTPUT instrumented-system
ENDFOR

ENDPROCESS


PROCESS: 2.3, Run Test

;PROCESS SPECIFICATION

IF test-directive CONTAINS "load-saved-state" THEN
   RETRIEVE saved-state FROM saved-state-data-base
ENDIF
FOR EACH test-directive DO
   /* execute test directive */
ENDFOR

ENDPROCESS
```

3.5 Data Reduction/Analysis Phase

The Data Reduction/Analysis Phase is shown in Figure 3-6. For failure effects analysis or conventional performance analysis, this phase consists mostly of grouping and analyzing collected data to determine actions, trends, etc. For reliability analysis, this phase consists of data reduction and statistical analysis, followed by the use of the results in a reliability model of the system.

3.6 Data Dictionary

The Data Dictionary follows Figure 3-6 and defines all terms used in the data flow diagrams as well as the mini-specs. For the Data Dictionary, the following symbols indicate:

1)	=	is composed of
2)	( )	optional item
3)	[a b c]	alternative items
4)	$n\{\}^{m}$	iterations of with optional lower (n) and upper (m) limits
5	+	and

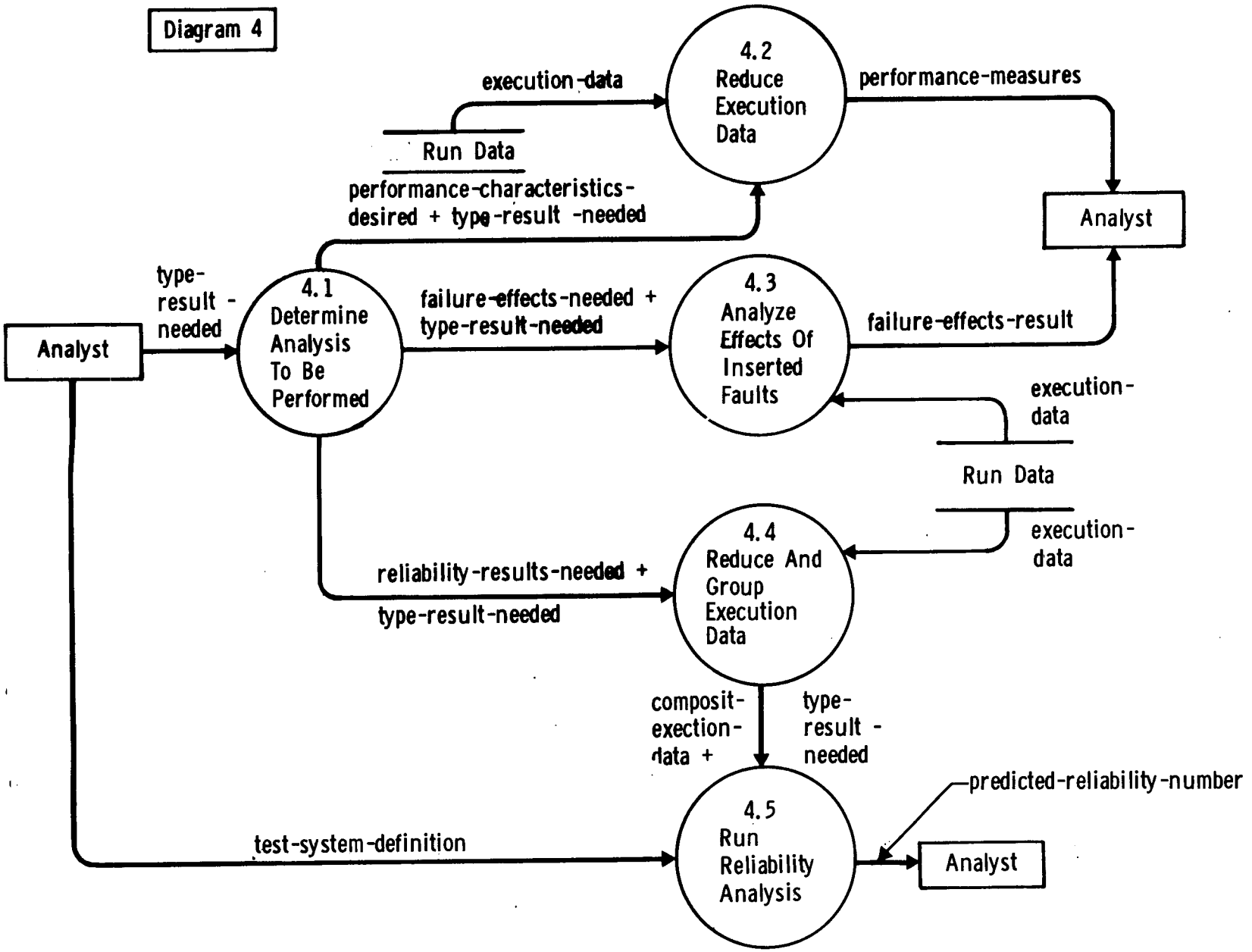


Figure 3-6 Perform Analysis

```
PROCESS: 4.1, Determine Analysis to Perform

;PROCESS SPECIFICATION

IF type-result-needed CONTAINS reliability-number THEN
   OUTPUT reliability-results-needed
ELSE
   IF type-result-needed CONTAINS failure-effects-analysis THEN
      OUTPUT failure-effects-needed
   ELSE
      IF type-result-needed CONTAINS performance-characteristic THEN
         OUTPUT performance-characteristics-desired
      ENDIF
   ENDIF
ENDIF

ENDPROCESS


PROCESS: 4.2, Reduce Execution Data

;PROCESS SPECIFICATION

ASSEMBLE execution-data USING performance-characteristics-desired
EXTRACT performance-measure

ENDPROCESS


PROCESS: 4.3, Analyze Effects of Inserted Faults

;PROCESS SPECIFICATION

FOR EACH faults-to-be-inserted IN execution-data DO
   DETERMINE (effect-of-fault AND propogation-of-fault) USING execution-data
   OUTPUT failure-effects-result
ENDFOR

ENDPROCESS


PROCESS: 4.4, Reduce and Group Execution Data

;PROCESS SPECIFICATION

ASSEMBLE execution-data USING specific-system-portion-of-interest
CHANGE execution-data TO composite-execution-data

ENDPROCESS
```

```
PROCESS: 4.5, Run Reliability Analysis

;PROCESS SPECIFICATION

/* Run reliability model using composite-execution-data */
CALCULATE predicted-reliability-number USING (composite-execution-data +
                                              confidence-level-desired)

ENDPROCESS
```

```
actuator-description = SELF_DEFINING /* description of actuators */
analysis-result = [performance-measure | reliability-number |
                  failure-effects-result]
block-functional-model = SELF_DEFINING /* description of the behavior of
                                          each functional block */
block-gate-model = SELF_DEFINING /* machine readable version of system logic
                                    diagrams broken into functional
                                    blocks */
block-gate-table = {gate-info}
block-gate-tables = {block-gate-table}
block-number = SELF_DEFINING  /* id of the block this gate is in */
boundry-information = SELF_DEFINING /* list of inputs and outputs to system */
code-generation-description = op-code-information + instruction-formats
composite-execution-data = {execution-data}
confidence-level = number
confidence-level-desired = percentage
configured-system = executable-environmental-model + gate-level-model +
                    functional-level-model + boundry-information +
                    loadable-software
current-gate-value = ["0" | "1" | "undefined" | "tri-state"]
data-desired = [number-of-samples-necessary + type-data-necessary +
                confidence-level-desired + type-of-failure-desired +
                {desired-failure-distribution} | type-data-necessary] +
                specific-system-portion-of-interest
data-recording-directive = data-recording-point + data-to-be-gathered +
                [time-interval | time | system-significant-event] +
                 output-device + output-format
data-recording-directives = {data-recording-directive}
data-recording-point = ["symbol" | "target memory location"]
data-recording-points = {data-recording-point}
data-to-be-gathered = SELF_DEFINING /* this item left unspecified since it
                                       could be wide range of possibilities,
                                       ranging from modeled items to actual
                                       items in the modeling machine */
desired-failure-distribution = probability-distribution
desired-performance-information = SELF_DEFINING /* this is the performance
                                  characteristic which we need to ascertain.
                                  Since the possibilities are numerous, this
                                  definition is not constrained. */
duration-of-fault = number
effect-of-fault = [stuck-at-fault | transient-fault]
environmental-model-description = {sensor-description} + {actuator-description}
                                  + {output-device-description} +
                                  {interconnection-description}
environmental-model-directive = initial-value + range-limits
environmental-model-directives = {environmental-model-directive}
environmental-model-performance = time + sensor-state
environmental-simulation-code = machine-object-code
environmental-symbol-table = symbol-table
event-identifier = ["sensor out of bounds" | "machine parameter out of bounds" |
                           system-significant-event]
```

```
executable-environmental-model = environmental-simulation-code +
                    environmental-symbol-table
execution-data = 1{run-id + [reliability-sample-data | performance-sample-data
                    | failure-effects-sample-data]}
execution-time = number
external-input = SELF_DEFINING /* this is an input from the system from the
                                  outside world. No restrictions are placed on
                                  its form or contents  */
external-interfaces = {external-input} + {external-output}
external-output = SELF_DEFINING /* this is an output to the outside world.
                                  No restrictions are placed on its form or
                                  content.    */
failure-effects-analysis = "failure effects needed" +
              specific-system-portion-of-interest + type-of-failure-desired
failure-effects-needed = specific-system-portion-of-interest +
              type-of-failure-desired
failure-effects-result = {faults-to-be-inserted + propogation-of-fault}
failure-effects-sample-data = 1{initial-state-data + 1{faults-to-be-inserted +
                                  {gate-behavior-data}}}
failure-mode-of-interest = failure-effects-analysis + faults-to-be-inserted
fault-insertion = "fault inserted" + faults-to-be-inserted
faults-to-be-inserted = location-of-fault + time-of-fault + effect-of-fault
              + duration-of-fault
functional-block = subsystem + internal-interfaces
functional-blocks = {functional-block}
functional-element-performance = SELF_DEFINING /* performance measures of some
                                                  portion of the system. This
                                                  item is so variable, it is not
                                                  specified in detail */
functional-level-model = functional-level-simulation-code +
              functional-level-symbol-table
functional-level-simulation-code = machine-object-code
functional-level-symbol-table = symbol-table
gate-behavior = last-gate-value + current-gate-value
gate-behavior-data = gate-id + machine-cycle + gate-behavior
gate-id = block-number + gate-number
gate-info = gate-state-info + {gate-output}
gate-interconnection-table = {gate-id + {gate-id}}
gate-level-model = {block-gate-table} + {gate-interconnection-table} +
              gate-symbol-table
gate-number = SELF_DEFINING /* id of this gate within its block */
gate-output = SELF_DEFINING /* pointer to one of this gates outputs */
gate-performance = {gate-behavior-data}
gate-state-info = gate-type + gate-value
gate-symbol-table = symbol-table
gate-type = ["AND" | "OR" | " NAND" | "NOR" | "INVERT" | "XOR" | "FLIP-FLOP"]
gate-value = ["0" | "1" | "undefined"   "tri-state"]
initial-state-data = time + {external-input} + {external-output} +
                     {sensor-state} + {internal-state}
initial-value = SELF_DEFINING
insertion-point = machine-object-code-location
```

```
insertion-points = {insertion-point}
instruction-formats = SELF_DEFINING /* information concerning addressing modes,
                                       bit patterns, etc as needed by the
                                       code generator */
instrumentation-point = machine-object-code-location
instrumentation-points = {instrumentation-point}
instrumented-system = {machine-object-code} + {data-recording-points} +
                {instrumentation-points}
interconnection-description = SELF_DEFINING /* description of how sensors,
                                               actuators, output devices are connected
                                               to the test system */
interface-behavior-model = SELF_DEFINING /* list of interconnections between
                                            functional blocks */
internal-interfaces = SELF_DEFINING /* connectiona between blocks */
internal-state = {machine-state}
interrupt = SELF_DEFINING
last-gate-value = ["0" | "1" | "undefined" | "tri-state"]
loadable-software = {machine-object-code + symbol-table}
location-of-fault = gate-id
lower-limit = SELF_DEFINING
machine-cycle = SELF_DEFINING /* id of the current machine cycle */
machine-object-code = SELF_DEFINING
machine-object-code-location = number
machine-state = SELF_DEFINING /* this is the state of the computer, including
                                 registers, memory, mode and any other
                                 parameters necessary to describe the current
                                 status of the machine itself */
model-information = ["gate-level-model-needed" | "model-subdivision-needed"
                    | "monolithic-model-needed"]
model-type-needed = "functional-model-needed" + ("gate-level-model-needed")
number = SELF_DEFINING
number-of-samples-necessary = number
op-code-information = SELF_DEFINING /* information concerning op codes as
                                       needed by the code generator */
output-device = ["disk" | "tape" | "console" | "line printer"]
output-device-description = SELF_DEFINING /* description of any other system
                                             output devices */
output-format = ["decimal" | "octal" | "hexidecimal" | "binary" |
                 "unformatted"]
percentage = number
performance-characteristic = "performance information needed" +
                desired-performance-information
performance-characteristics-desired = specific-system-portion-of-interest
performance-measure = SELF_DEFINING /* this will depend on the type of measure
                                       desired. this is highly variable so
                                       no enumeration is given here */
performance-sample-data = 1{initial-state-data + {significant-event-data}}
predicted-reliability-number = number + confidence-level
probability-distribution = SELF_DEFINING
propogation-of-fault = {gate-behavior-data}
```

```
range-limits = upper-limit + lower-limit
reliability-number = "reliability number needed" + confidence-level-desired
reliability-results-needed = confidence-level-desired
reliability-sample-data = 1{sample-number + initial-state-data +
                              {significant-event-data}}
run-id = number
sample-number = number
saved-state = [configured-system | instrumented-system] + time +
              {external-interfaces} + {sensor-state} + {internal-state}
saved-state-data-base = {saved-state}
sensor-description = SELF_DEFINING /* description of what sensor is and how it
                                      behaves. May be text */
sensor-state = SELF_DEFINING /* this is the current state of the sensor as
                                defined by some parameters such as orientation,
                                or by its output values */
significant-event-data = time + {external-input} + {external-output} +
                         {sensor-state} + {internal-state} + event-identifier
specific-system-portion-of-interest = functional-block
stuck-at-fault = [stuck-at-one-fault | stuck-at-zero-fault |
                  stuck-at-indeterminate-fault]
stuck-at-indeterminate-fault = SELF_DEFINING
stuck-at-one-fault = SELF_DEFINING
stuck-at-zero-fault = SELF_DEFINING
subsystem = SELF_DEFINING /* any reasonable chunk of the system which can be
                             isolated as an identifiable piece */
symbol-table = insertion-points + instrumentation-points
               /* + a bunch of other stuff */
system-block-diagram = SELF_DEFINING /* block diagram of the system of interest
                                        showing major components and their
                                        interfaces */
system-block-diagrams = {system-block-diagram}
system-boundries = { system-boundry}
system-boundry = {functional-block} + {internal-interfaces} +
                 external-interfaces
system-environmental-description = SELF_DEFINING /* description of the behavior
                                                    of the system external
                                                    environment including all
                                                    input and output */
system-functional-description = SELF_DEFINING /* description of the functional
                                                 level behavior of the system,
                                                 including instruction fetch and
                                                 decode of the computer(s) */
system-logic-diagram = SELF_DEFINING
system-logic-diagrams = {system-logic-diagram}
system-model = executable-environmental-model + functional-level-model +
               (gate-level-model) + boundry-information
system-significant-event = [interrupt | trap | fault-insertion]
test-conduct-directive = initial-state-data + execution-time + sample-number
test-directive = {faults-to-be-inserted} + {environmental-model-directive}
               + {test-conduct-directive}
```

```
test-directives = {test-directive}
test-software = SELF_DEFINING /* source software for the system under test */
test-system-definition = system-environmental-description +
                (system-logic-diagrams) + system-functional-description +
                test-software + system-block-diagrams
time = number
time-interval = number
time-of-fault = number
transient-fault = SELF_DEFINING
trap = SELF_DEFINING /* this is the occurrance of a system trap inserted for
                        the purposes of recording data or some such reason */
trap-insertion = SELF_DEFINING
type-data-necessary = (gate-performance) + (functional-element-performance) +
                (environmental-model-performance)
type-of-failure-desired = [stuck-at-fault | transient-fault]
type-result-needed = [performance-characteristic | failure-mode-of-interest
                | reliability-number] + specific-system-portion-of-interest
upper-limit = SELF_DEFINING
```

4.0 System Specification

4.1 General System Configuration

This specification describes the requirements for the digital avionics design and reliability analyzer. This facility consists of two major hardware itemsas shown in Figure 4-1, a general purpose computer providing user support and interface, simulation, and numerous other pieces of software; and an emulation computer to provide either gate level emulation or general instruction level hardware emulation. These two computers are interfaced for synchronization and data transfer. The software for the facility is shown in diagram 4.2. Of the five major components only a small part of the general purpose support software, the model building software and the test execution software run on the emulation computer. The major portion of the software runs on the general purpose computer.

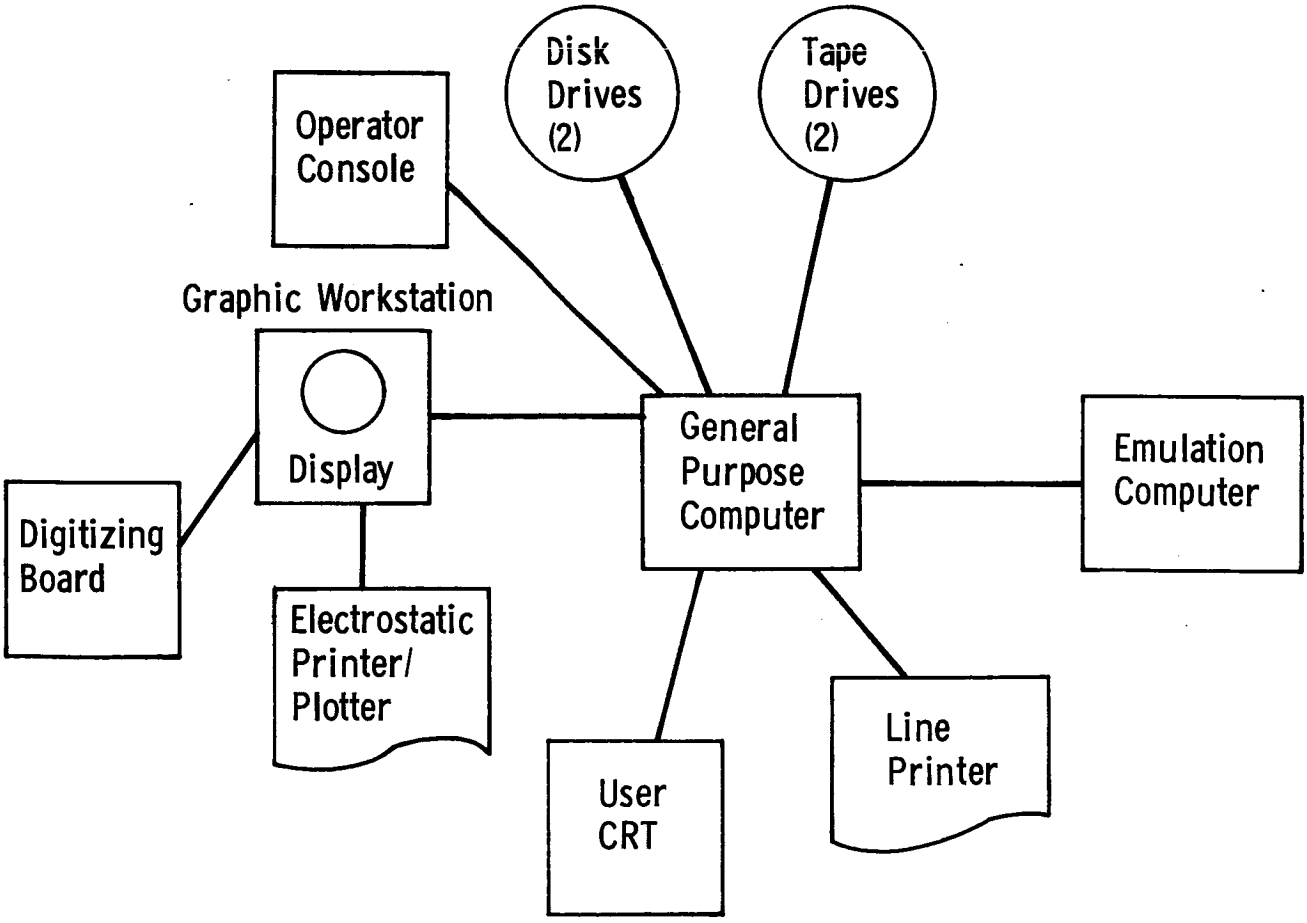


Figure 4-1 System Hardware Components

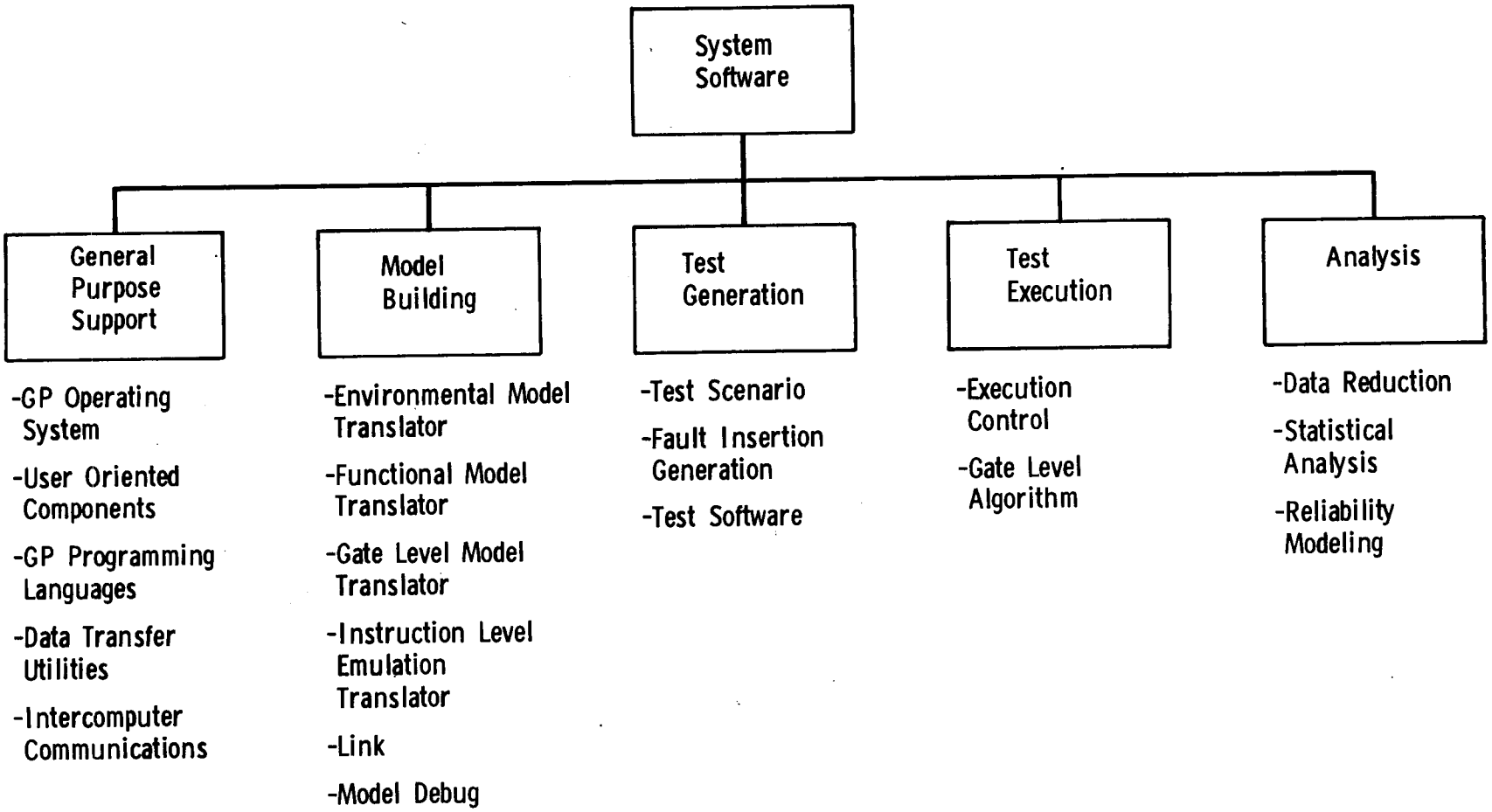


Figure 4-2 System Software Components

4.2 Hardware Configuration

The system shall consist of two cooperating machines connected via an interface. These machines shall be:

1) A general purpose computer providing user interface; software support such as editors, assemblers, compilers, simulation support; and analysis support.

2) An emulation computer supporting emulations ranging from gate level to instruction level.

4.2.1 General Purpose Machine

4.2.1.1 Central Processor

4.2.1.1.1 The system shall have a real-time clock (interval timer) for use by the operating system

4.2.1.1.2 Machine hardware instructions shall include integer, single and double precision floating point, packed-decimal, character string manipulation, bit shifting and rotating, and logical instructions.

4.2.1.1.3 Hardware fault detection shall be provided, i.e., detection of divide by zero, exponent overflow, and exponent underflow.

4.2.1.1.4 The system shall detect a power failure or fluctuation and have the capacity to provide for an orderly system shutdown. Upon re-establishment of stable power, automatic restart of the system must be provided for. This requirement may be met by battery back-up to maintain proposed MOS (metal oxide semiconductor) memory allowing for operator notification and intervention. The system must be maintained for a long enough period to permit any necessary steps to be accomplished to allow for restart of the system and user programs.

4.2.1.1.5 The architecture of the system shall be based on a computer with effective addressing, register size, and interger arithmetics of at least sixteen (16) bits.

4.2.1.1.6 The general purpose computer shall have the speed and power necessary to execute the environmental model and the functional level model specified in 4.3.2.1 in the normal operating mode, cooperating with each other, at a slow down of not more than 3000 times real time.

4.2.1.2 Memory

4.2.1.2.1 The memory requirements stated are in terms of bytes. A byte is defined as the alphanumeric character oriented unit of measure composed of a least eight (8) bits. Manufacturers whose internal architecture is such that they normally operate with less than 8 bit bytes must adjust their bytes or words of memory proposed to reflect the 8 bit requirement. Memory single word size must be at least sixteen (16) bits available to user programs.

4.2.1.2.2 The initial configuration must be a minimum of one-half (1/2) million bytes of main memory. The system architecture shall not preclude a single user program from utilizing the full complement of main memory beyond the residency requirement of the operating system and related software.

Both hardware and software shall support two (2) million bytes of physical memory for expansion purposes.

4.2.1.2.3 Areas or regions of memory shall be memory protected to facilitate the protection of the operating system and individual user programs. This requirement may be met by any combination of hardware and/or software features.

4.2.1.2.4 Single bit fault correction and multiple bit fault detection shall be provided.

All detected memory faults shall be logged by the system. This log shall be accessible by either a vendor, customer engineer, and/or government personnel.

4.2.1.2.5 The rationale for the one-half (1/2) million bytes of main memory is as follows:

1) Traditionally, interactive graphics systems tend to be complex and to require significant amounts of memory to operate effectively. The interpretive graphic subsystem is only a small portion of the total system and will undoubtedly have to operate concurrently periodically with other tasks. Even if operating by itself, it is quite conceivable that once in a production mode that multiple digitizing stations will be required.

2) As detailed system design has not been completed, it is difficult to predict with an accuracy the ultimate memory requirement of the test execution software. The following items will need to be memory resident for the test execution and in total will be significant in terms of memory required:

   a. Fault tolerant target machine object code.

   b. Compiled hardware description code for the target machine.

   c. Actuators/sensors values and associated bound limits.

   d. Fault data being introduced.

The fault tolerant target machines will be complex in terms of having redundant hardware components and significant associated control/management software.

3) With anticipated run times of test execution software to be in terms of hours or days, throughput can not be significantly degraded due to excessive page thrashing and/or overlay roll-in and roll-out. The requirement for physical memory to be expandable to two (2) million bytes is to keep execution times within reason as the fault tolerant systems under study become more complex.

4.2.1.2.6 Memory allocation shall be dynamically allocated with the ability to support at least four (4) interactive devices concurrently at installation time and expandable to eight (8). A minimum of two (2) batch jobs must run concurrently with the interactive users. The environment is to be that of true multiprogramming, i.e., a fixed partition foreground/background environment specifically shall not be permitted.

4.2.1.3 Disk Storage

4.2.1.3.1 Five-hundred (500) million 8 bit bytes of removable and interchangeable formatted disk storage shall be available to the users of the system. Disk storage required for system software is in addition to this requirement. This disk space for the system shall be expandable by a factor of two (2).

4.2.1.3.2 Average access time including latency and seek time, shall be 55 milliseconds or faster. The transfer rate shall not be less than 800,000 8-bit bytes per second.

4.2.1.3.3 The vendor shall provide an initial complete set of recording media, as well as a complete backup set, both containing no more than 0.01% unnacceptable sectors per unit.

4.2.1.3.4 A minimum of two (2) physical drives are required.

4.2.1.3.5 The five-hundred (500) million 8-bit bytes of removable and interchangeable formatted disk storage for the user is considered justified for the following reasons:

1) Disk I/O spooling area for local print output. It is anticipated that the various report products shall be maintained on disk for several work days during their review, the rationale being to save computer run time in the event that additional copies are required for further study and distribution.

2) Provision for multiple files of gate level logic diagrams with associated legend. These files represent the various portions of the target fault-tolerant computer system under evaluation. Different portions of the target fault-tolerant computer system will be at different stages of the capturing and editing of gate level logic diagrams via the interactive graphics subsystem.

3) Program source code library.

4) Program object code library.

5) Multiple gate queueing structure tables in emulator computer compatable format.

6) Multiple fault data files.

7) Multiple files of the initial conditions and bound limits of the avionic actuators and sensors of the fault tolerant systems under study.

8) Multiple hardware configuration descriptions defining various fault tolerant system options.

9) Library of procedure files and parameter files.

10) Data files associated with mathmatical and statistical analysis routines.

11) As the mechanical/electronic nature of disk drives require frequent maintenance, the requirement of two physical spindles was specified to allow some work to continue when one drive in unavailable. Admittedly, the capability will require careful organization of the disk files.

12) Further, utilizing a large capacity disk drive allows achievement of economy of scale. For example, a calculation revealed that, for one vendor, going from a medium to a large capacity drive resulted in a 162% increase in capacity for a 76% increase in cost.

13) Disk space is required for the recording of data during the execution of test software (4.3.4). The approach taken of recording information only when out of limits conditions occur (4.3.4.4.3) is a compromise over what the run data recording requirements could be. The calculation provided here is an example of what the run storage requirements would be if recording of data were to be done for each target machine simulated/emulated cycle.

Assumptions:
6,000 gates
15% gate state changes
1μs target machine cycle time
1,000 samples per run.

Calculations:
900 (15% of 6,000) gate state changes per cycle
900,000,000 gate changes per sample
9000,000,000,000 gate changes per run
With 3 gate changes recorded per
32 bit word, 300,000,000,000 words of disk required.
1,200,000,000,000 bytes of disk required
or if on magnetic tape
With 4,000 charactor tape blocks
300,000,000 blocks at 3 inches of tape each is 900,000,000 inches of tape
900,000,000 - 28,800 = 31,250 2400 foot reels

4.2.1.4 I/O Devices

4.2.1.4.1 Tape Drivers

4.2.1.4.1.1 Two (2) read/write nine track 1600 CPI phase encoded tape drives of not less than 75 IPS read/write speed or less than 120,000 bytes per second peak transfer rate shall be provided.

4.2.1.4.1.2 The tape units shall provide for read-after-write check feature.

4.2.1.4.1.3 The tape units shall handle up to 2400 foot reel size.

4.2.1.4.1.4 The tape units shall be of the vacuum chamber type. Mechanical feed arms are not permitted.

4.2.1.4.2 Line Printer

4.2.1.4.2.1 One (1) impact type printer shall be provided. The printer shall have no fewer than 132 print positions. The ASCII character set of 95 characters shall be employed. The proposed printer shall be able to line space at 6 and 8 lines per inch, vertically. The printer must provide standard horizontal spacing of ten characters to the inch.

4.2.1.4.2.2 The throughput requirement is minimally 600 lines per minute when printing full 132 character lines consisting of the 95 printable character set.

4.2.1.4.2.3 The system shall be upgradable to a configuration of two (2) printers meeting these specifications.

4.2.1.4.3 Operator Console

4.2.1.4.3.1 The system operating console shall provide for hard copy output. The console must be of rugged construction capable of withstanding heavy use, i.e., continuous use during operating hours. This requirement would not preclude a printing unit operating as a slave to a CRT operator console.

4.2.1.4.4 Telecommunications Hardware

Telecommunication hardware shall be provided to handle data exchange and its associated line disciplines between local terminals and the host computing system. Attached terminals will be used for time sharing, inquiry/response, local graphics and local plotting. The system shall be able to handle half and full duplex lines concurrently. Circuit disciplines in general shall include at the minimum start/stop half duplex and full duplex asynchronous transmission.

4.2.1.4.4.1 All telecommunications hardware supplied by the vendor shall conform to the Electronic Industries Association Standard RS 449. The government has adopted RS-449 as Federal Standard 1031, which became mandatory for all procurements by federal agencies starting June 1, 1980. EIA Standards RS-449, RS-422, and RS-423 are intended to gradually replace RS-232-C. Telecommunication hardware which conforms to the new standards shall be provided. The vendor's proposal must state how existing terminals which conform to RS-232-C will be accomodated.

4.2.1.4.4.2 The telecommunications hardware/software shall support the Teletype (TTY) start-stop asynchronous communications. The emulators/simulators are intended for use with various ADP vendors, so the proposed emulation/ simulation shall not be specifically designed for any particular vendor.

4.2.1.4.4.3 The initial four (4) communications ports (See A.2.2.5) will be utilized for some combination of alphanumeric CRT(s), graphic display, digitizer board, and electrostatic plotter. The proposed system must be upgradable to eight (8) communication ports.

4.2.1.4.4.4 One (1) interactive user CRT terminal is to be provided by the vendor. The unit will be locally attached to the CPU operating at the speed of 1200 BPS or faster. The physical connection will not exceed the industry standard of fifty feet. This unit will be utilized by the software program specified under 4.3. The following are minimum specifications to be met:

1) 80 character line width

2) 24 vertical lines

3) Fill-in-the-form capability with the form stored in the background and variable information entered in the foreground. Once the form has been loaded to the CRT memory, it will be utilized for a series of transactions without need for retransmission from the computer. Only the variable information is to be transmitted to the computer during the data entry process.

4) Normal and reverse video

5) Double intensity

6) Blinking

7) Underlining

4.2.1.4.5 General Purpose Computer Emulation Computer Interface

4.2.1.4.5.1 The contractor shall provide any necessary hardware to interconnect the general purpose computer to the emulation computer. (Related software is specified under 4.3).

4.2.1.4.5.2 Data transfers between the two computers will consist of the following:

4.2.1.4.5.2.1 For reliability analyses data gathering, the transfers will include:

1) At the start of the run, the general purpose computer will load the emulation computer control memory with the gate level emulation algorithm and the emulation computer primary memory with the gate tables for the specific system portion of interest.

2) During the run at each fault insertion time, the gate tables within the emulation computer primary memory will be updated to reflect the inserted fault by the general purpose computer.

3) During the run, for each machine cycle, the inputs to the block being emulated at the gate level will be transferred from the general purpose machine to the emulation machine and the outputs of the block will be transferred back from the emulation machine to the general purpose machine. The quantity of data transferred depends on the degree of interconnection between the emulated block and the rest of the system.

4.2.1.4.5.2.2 For failure effects analysis, the data transfers will be the same as specified in reliability analysis data gathering. In addition, at the end of each cycle, the new state of each changed gate may potentially be transferred from the emulation computer back to the general purpose computer.

4.2.1.4.5.2.3 For standard emulation purposes, data transfers will be as follows:

1) At the start of the run, the emulation machine control and primary memory will be loaded with the appropriate software by the general purpose machine.

2) During the run, input and output data from the environmental simulation to the emulated machine and back will be transferred at appropriate times.

3) Additional data concerning the state of items in the emulated machine may potentially be transferred back to the general purpose machine for performance evaluation purposes.

4.2.1.4.5.2.4 The speed of the computer-computer interface for data transfer shall be sufficiently fast so that the predominant amount of time in the reliability analysis data gathering experiments will be time for simulation/emulation of the system rather than for data transfer.

4.2.1.5 Interactive Graphics Subsystem

4.2.1.5.1 The contractor shall provide the necessary hardware to capture and validate gate level logic diagrams. (Related software is specified under 4.3)

4.2.1.5.2 The graphics workstation is to be made up of the following components:

1) Digitizing Board with cross-hair cursor. The digitizing surface must be large enough to handle logic diagrams up to standard size E (34" X 44"). The gantry style digitizer is preferred, but is not mandatory. The logic symbols are not to be digitized in detail. The symbol type is to be selected from a menu and the symbol position is to be recorded via the cursor. With this approach, a digitizer board with minimal accuracy, resolution, and repeatability may be utilized. A resolution of 100 points per inch is adequate. The working surface shall have both tilt and height control.

2) Graphics CRT with alphanumeric keyboard. The minimum screen size shall be 19". This requirement may be met by a single raster scan type graphic CRT with the capability of a reference drawing being flashed onto the screen from which a zoom-in area may be selected. The requirement may also be met via two storage tube type graphic CRT's. A reference drawing would be displayed on one CRT while zoom-in areas are displayed on the second CRT.

3) Electrostatic plotter with a roll paper width of 36". The unit shall have a resolution of 100 points per inch. The electrostatic plotter will be used primarily for quick turnaround images for validating plots against the original input document, i.e., gate level logic diagrams.

4.2.2 Emulation Computer

4.2.2.1 CPU Architecture - The emulation computer shall be user microprogrammable. The microcode shall provide control over primitive functions within the machine (e.g. connection of registers to busses, ALU operations, etc) and shall provide the capability for parallel operations within a microword.

4.2.2.1.1 Microcode containing the NASA Langley gate-level algorithm or similar algorithm must be programmed into the emulator. Due to the stringent speed requirements for processing such an algorithm, the microcode must perform multiple operations in parallel.

4.2.2.1.2 Each gate being processed is described by a gate information word of eight or more bits. This word is also the address to which control is transferred in micro store, thus micro store must be sufficient to handle all locations addressed.

4.2.2.2 Memory

4.2.2.2.1 Microprogram Memory - Sufficient microprogram memory shall be provided to accomodate a table-driven gate level emulation algorithm. Requirements of the algorithm are detailed in paragraph 4.3.4.7. As a minimum, at least 1K words of microprogram memory shall be provided.

4.2.2.2.2 Primary Memory - Sufficient primary memory shall be provided to contain the gate level tables required by the emulation algorithm. These tables shall accomodate at least 5000 gates with an average gate fan-out of 2. As a minimum, at least 32K words of primary memory shall be provided.

4.2.2.3 The selection of the micro code to be executed shall be via a "vector" type mechanism. That is, some combination of bits in a word containing gate status shall provide the address of the microinstruction to be executed. Such a mechanism precludes the necessity of testing individual bits to determine the action to take for a particular gate.

4.2.2.4 The emulation computer shall also be useful for instruction level emulation of digital devices. The characteristics of the machine shall be such that it will accomodate such emulation.

4.2.2.5 The emulation computer shall be interfaced to the general purpose computer for data transfer and for software level synchronization of cooperating, parallel simulations and emulations in the two machines. Data transfers expected are defined in 4.2.1.4.5.

4.2.2.6 The emulation computer shall have the speed and power necessary to execute the gate level emulation, in the normal operating node, for 6000 gates for 0.1 seconds of emulated time in 5 minutes or less of real time. The cycle time of the emulated system for this timing figure shall be 1 microsecond, the average gate fanout shall be 2, and in any one cycle, 5% of the gates will change value, on the average.

4.3 Software Configuration

The software consists of five major pieces. These pieces are:

1) General purpose support software

2) Model building software

3) Test generation software

4) Test execution software

5) Analysis software

The software, with exception of some of the test execution software, some of the general purpose support software, and some of the model building software shall run on the general purpose machine.

4.3.1 General Purpose Support Software

4.3.1.1 General Purpose Machine Operating System

4.3.1.1.1 The system shall feature a single, fully implemented operating system that integrates all the hardware and software that comprise the system. The operating system shall be generally available in the market place. More specifically, all features and capabilities shall have been publicly and formally announced and operational prior to the offer submission deadline.

4.3.1.1.2 It is anticipated that the primary mode of operation will be a single operator performing a single task. Examples would be a single graphic station capturing a logic diagram or a simulation/emulation job running alone in the system. However, the architecture of the system shall not preclude the concurrent processing of a simulation/emulation run with the digitization process. Nor should the architecture preclude the addition of a second graphic work station in the future to operate concurrently with the original graphic work station.

Allocation of resources to tasks shall be performed as automatically as possible. All the software items specified throughout this document shall be able to operate concurrently with any and all others, except for restrictions such as momentary unavailability of an equipment resource.

4.3.1.1.3 The operating system shall provide a dynamic environment. That is, memory management shall be done in such a manner that all concurrent running jobs in total may require more memory than what is physically available. The addition of more physical memory would improve the system's performance. This capability shall be provided without requiring the programming staff to define overlays.

4.3.1.1.4 The operating system shall have the ability to produce and retain in mass storage for later processing, resource utilization data pertinent to each task performed. The resource data produced shall include most of the following by user account/charge:

1) Number of lines printed

2) Central processing unit usage

3) Input/output usage

4) Remote terminal connect time or traffic statistics

5) Actual memory used

6) Amount of mass storage used

Simplified measuring units such as the aggregate of the items above, shall be reversible to the individual component level.

4.3.1.1.5 The operating system shall operate the following basic job origin tasks concurrently:

1) Interactive

2) Local batch

4.3.1.1.6 The operating system shall provide for I/O spooling. Spooling of local print output shall be provided for. Spooling is defined here as providing a temporary file that will act as a buffer for spontaneous input or output of data and thereby reduce impacts to executing programs waiting for I/O services. Direct, i.e., non-spooled, I/O shall also be available for time-critical transmissions.

4.3.1.1.7 Terminal users of the system shall be able to communicate with the system operator via terminals and vice versa via the operator control console. This is required because terminal users may or may not reside in the same room as the system.

4.3.1.1.8 Interactive batch job submittal from time-sharing devices shall be provided. The user shall be permitted to save files on either magnetic tape or mass storage disk.

4.3.1.1.9 A terminal user shall be able to determine the status of a batch job from a time-sharing terminal.

4.3.1.1.10 The system response time to an interactive user system command shall not exceed an average of two (2) seconds. The absolute maximum response time shall not exceed thirty (30) seconds.

4.3.1.1.11 The system must provide for a job control language that allows the user to override system defaults and parameters pertinent to job management, job scheduling and data management. This provision shall provide control over job priorities; job termination options; programmatic steps within a job stream; job dispatching and execution, etc.

4.3.1.1.12 Security and system authorization. The system must limit access to any and all installation resources, including files and data contained therein. This facility will only allow processes to those users that are pre-defined as authorized for access. Read and write permits must be features within the data authorization scheme.

4.3.1.1.13 The operating system shall be considered to be state-of-the art. That is, the operating system shall have been designed, developed and implemented to support an environment of concurrent interactive and batch jobs. The system being specified in this document is to be utilized in the evaluation and testing of fault tolerant airborne avionic computers of the future. When consideration is given to this fact and the fact that the environment is one of new technological development, it is prudent and reasonable that only the best available resources and tools should be made available for the project.

4.3.1.2 User Oriented System Software Components

4.3.1.2.1 A file editor shall be provided with the following minimum capabilities:

1) With the exception of binary object files and files written by FORTRAN as unformatted, be able to manipulate any and all files used by the system.

2) Must be available interactively and optionally be available through the batch mode.

3) Must contain, as a minimum, the following, or equivalent capabilities:

   1. Replace String

   2. Change line

   3. Delete

   4. Print/List

   5. Search (forward and backward)

   6. Insert

   7. Add

4) Must provide the user with the view that his whole file is immediately available to him, that is, he must not have to specifically fill and empty the current edit buffer.

4.3.1.3 General Purpose Programming Languages

4.3.1.3.1 A FORTRAN compiler that minimally meets the ANSI X.3.9-1966 specifications shall be provided. The delivered compiler must be stable and thoroughly debugged.

4.3.1.3.2 An ASSEMBLER or hardware level compiler shall be provided which possesses features not available in the high level programming languages required under 4.3.1.3.1 and 4.3.2.2. Bit and character level manipulation, privileged instructions, register referencing, and branching based on hardware conditions shall be provided.

4.3.2.1 The relationships of the various models shall be as follows:

4.3.2.1.1 The environmental simulation shall execute in the general purpose machine and shall simulate the effects of the systems, sensors and activators which interface to the digital avionics computer(s). This would include such items as attitude and rate sensors, attitude control activators, etc. This model shall produce the identical effect as if the avionics computer(s) were connected to actual devices in a real system.

4.3.2.1.2 Functional level machine simulation shall provide a model of the behavior of the avionics computer(s). This simulation shall be at the instruction level of the computer such that actual software may be executed by the simulation with results identical to the real avionics computer(s). This model shall interact with the environmental model, reacting to the inputs provided by that model and producing the appropriate outputs to that model. This model shall also interact with the gate level emulation(s) which are active by providing the appropriate inputs and receiving the outputs of the gate level emulation(s).

4.3.2.1.3 The gate level emulation shall provide a model of the behavior of a portion of the digital avionics computer(s) at the gate level. It shall interact with the functional level simulation by receiving inputs from that simulation and by providing appropriate outputs to the simulation. This model will correctly propagate inserted faults to its outputs.

4.3.2.1.4 The instruction level machine emulation is intended to interact only with an environmental simulation. This model shall provide the capability to emulate, on the emulation machine, a complement of digital hardware at the instruction level. This is intended for gathering data concerning performance and not for failure effects or reliability evaluation.

4.3.2.2 Model Description Translators

4.3.2.2.1 Environmental Model Translator

4.3.2.2.1.1 A translator shall be provided which will translate an environmental description into executable simulation code on the general purpose machine. The translator shall accomodate descriptions of outputs, inputs, limits, etc., for sensors, activators and items external to the avionics computer(s). The translator shall accomodate tagging items of interest for later checking on limits during the simulation execution.

4.3.2.2.1.1 The environmental model execution code shall interface to and provide input and output for the functional level computer simulator.

4.3.2.2.2 Functional Model Translator

4.3.2.2.2.1 A translator shall be provided to translate a functional or instruction level description of one or more digital avionics computers into executable simulation code on the general purpose machine.

4.3.2.2.2.2 The functional model translator shall use a hardware description language which allows expression of the structure and behavior of digital systems.

4.3.2.2.2.2.1 The hardware description language shall provide for expression of timing and synchronization, both between internal elements and between the system being described and the external environment.

4.3.2.2.2.2.2 The hardware description language shall provide for description of the interface between the system being described and the external environment in terms of inputs and outputs. This description shall provide the tie to the executable code for the environmental model so the two models will work together.

4.3.2.2.2.2.3 The hardware description language shall allow the description of the system in terms of independent functional blocks and the interfaces between those blocks. The translator shall produce code which allows the replacement of the code for a functional block with something else which will provide the same inputs and accept the same outputs without modifying the model itself. This shall provide the link between the functional simulation model and the gate level emulation model.

4.3.2.2.3 Gate Level Model Translator

4.3.2.2.3.1 The gate level model translator shall translate system logic diagrams to the gate level tables needed by the gate level emulation algorithm.

4.3.2.2.3.2 The gate level model translator system shall include all necessary interactive graphics software to operate on the general purpose computer for capturing and validating logic diagrams. Standard logic symbols shall be used for:

1) Inverter

2) AND gate

3) OR gate

4) XOR (exclusive OR) gate

5) NAND (not AND) gate

6) NOR (not OR) gate

7) RS flip-flops

8) T flip-flops

9) D flip-flops

10) JK flip-flops

11) other logic devices.

The software capability to capture legend and associate the legend with each gate or device symbol shall be provided. The software shall provide for multiple logic diagram sheets for a single function to be emulated. That is, a single logic diagram up to 34" X 44" in size will not always represent an entire function to be emulated as a complete unit. Yet each separate sheet must be stored on disk as a subunit for output on the electrostatic plotter for the validation process. Off diagram linkages to other sheets must be provided for. A single "E" size drawing 34" X 44" will represent approximately 1500 gates.

4.3.2.2.3.3 The translator shall translate the logic diagrams captured via the graphics system to produce the gate level tables required by the gate level emulation algorithm on the emulation computer. Each functional block, corresponding to the functional blocks of the functional level simulation model, shall be in a separate table, identifiable with the corresponding functional level block.

4.3.2.2.3.4 A language translator shall also be provided which allows description of gates and their interconnections in a purely textual manner. The output of this translator shall be identical to and compatible with the graphics input translator.

4.3.2.2.4 Instruction Level Emulation Translator

4.3.2.2.4.1 The instruction level emulation translator shall provide the same functions as the functional model translator specified in 4.3.2.2.2 except that the executable code to which the description is translated shall be code for the emulation computer rather than the general purpose computer.

4.3.2.2.4.2 The instruction level emulation translator shall use a hardware description language but there is no requirement for partitioning into functional blocks. The translator shall provide the interfaces to the environmental model simulation running in the general purpose computer.

4.3.2.2.4.3 The instruction level emulation translator shall provide a code generation description output which may be input to a retargetable software translator such as a compiler or assembler which will translate the software to drive the described system. The description shall provide instruction formats, machine code descriptions and any other data necessary.

4.3.2.3 Link Software. Any software necessary to link the various models and allow them to communicate shall be provided.

4.3.2.4 Model Debug Packages

4.3.2.4.1 Debug packages shall be provided for each type of model.

4.3.2.4.2 The debug packages shall support interactive control and display of actual system parameters and modeled system components.

4.3.2.4.2.1 The debug packages shall support control of each model including as a minimum

1) Start

2) Stop

3) Single Step

4) Trace

5) Breakpoints (minimum of 16)

6) Item value change trace

7) Continue after break

8) Interactive modification of values

4.3.2.4.2.2 The debug package shall support display of both actual and modeled systems items including as a minimum:

1) memory

2) registers

3) emulated gates

4) external inputs and outputs (environmental simulation)

5) processor state

6) time

4.3.2.4.2.2.1 The items to be displayed shall be specifiable by the operator including display device and format.

4.3.2.4.2.2.2 Items shall be capable of being tagged for display in response to system events such as:

1) breakpoint

2) interrupt

3) user command

4) trace

5) single step

4.3.3 Test Generation Software

Software for developing test scenarios and fault insertion shall be provided.

4.3.3.1 Test Scenario Software

4.3.3.1.1 The test scenario software shall include the capability to specify a sequence of runs, perhaps with differing parameters, which will subsequently be run automatically by the system.

4.3.3.1.2 The test scenario software shall include the capability to specify initial values of all external inputs, simulated devices and internal state, including time, of the test system. This shall include the capability to load a configured system which has been previously stored on a storage device.

4.3.3.1.3 The test scenario software shall include the capability to specify specific data to be collected, the format of the data and the event in response to which the data shall be recorded.

4.3.3.2 Fault Insertion Generation

4.3.3.2.1 The fault insertion generation software shall provide the capability for either automated or manual generation of faults to be inserted in the gate level emulation.

4.3.3.2.2 The fault insertion generation software shall produce the following information concerning each fault to be inserted:

1) Gate identifier to receive the fault

2) The simulation/emulation run time at which the fault is to be applied in terms of sample number and fraction of time within the sample.

3) The duration of the fault

4) The fault state that is to be introduced, i.e., steady zero state, steady one state, intermittent zero state, or intermittent one state, or alternating between the zero and one state.

4.3.3.2.3 The manual fault generation software shall allow the analyst to specify all of the factors for each fault as given in 4.3.3.2.2.

4.3.3.2.4 The automated fault generation software shall allow the analyst to specify the following:

1) Number of faults to be generated

2) Specific system portion of interest or probability distribution of faults across the system

3) Probability distribution of faults over time for each sample

4) Probability distribution of type and duration of faults.

4.3.3.2.4.1 The automated fault generation software shall produce the data specified in 4.3.3.2.2 through the use of random number generators to provide the desired distributions.

4.3.3.2.5 The fault generation software shall produce the information specified in 4.3.3.2.2 in such a way that the simulation execution system will use it to insert faults at the specified time in the specified sample in the gate level emulation.

4.3.3.3 Test Driver Software Generation

4.3.3.3.1 The capability shall be provided for translating software for the target (emulated/simulated) machine on the general purpose computer. The translated software shall be used to drive the test system during the execution phase.

4.3.3.3.2 It is highly desirable that the translation process be entirely automated, taking as input the source code and a description of the machine for which code is to be generated and then producing object code for that machine. Translators which operate in this mode are often referred to as meta assemblers or meta compilers.

4.3.3.3.3 It is highly desirable to have a meta compiler delivered to satisfy this requirement. However, given the current state-of-the-art, a meta compiler with retargetable code generator is not available. As a minimum a meta assembler is required.

4.3.3.3.3.1 The meta assembler will have as one input a description of the instruction formats, operation codes and addressing modes of the machine for which code is to be generated. This input shall be produced from the hardware description language specified in 4.3.2.2.2, augmented as necessary for this particular task.

4.3.3.3.3.2 If a meta compiler is proposed, it shall have the same requirements as specified for the meta assembler in 4.3.3.3.3.1.

4.3.4 Test Execution Software

Software for executing the given test shall be provided. This software shall provide for the control of all the simulation and emulation models during a test.

4.3.4.1 The execution software shall provide for coordination of timing between the various models so that they are all synchronized in relation to simulated time.

4.3.4.2 The execution software shall also coordinate the execution of the simulations and emulations in the following manner for tests in which fault insertion is used.

4.3.4.2.1 The execution software shall execute the functional level model and the environmental model until the time at which a fault is to be inserted.

4.3.4.2.2 When the fault is to be inserted, the execution software shall cause the execution of the functional block in which the fault is inserted to switch from the functional level simulation to the gate level emulation with the inserted fault. The balance of the simulated system, without the faults, will continue at the functional level.

4.3.4.3 The execution software shall provide for the data recording specified under the test scenario software in reaction to the events specified.

4.3.4.4 For data reliability analysis collection, the following data collection shall be provided.

4.3.4.4.1 At the start of a sample, sample number, initial conditions (external inputs and outputs and internal system state) and all other pertinent information shall be recorded.

4.3.4.4.2 At the time of insertion of a fault, the sample number, system inputs and outputs and internal system state, and all the information concerning the fault shall be recorded.

4.3.4.4.3 At any time during the run, whenever any of the inputs or outputs exceed the limits specified under the test scenario software, the sample number, simulated time, inputs, outputs, internal system state, and value out of limits shall be recorded.

4.3.4.4.4 As the test execution software may very well run for hours or even days, it is absolutely mandatory that automatic check-point restart capability be provided.

4.3.4.5 The execution software shall operate without user intervention but shall allow the user to stop the execution and save the system state for later reload.

4.3.4.6 The execution software shall provide support for all of the model debug packages specified in 4.3.2.4. It shall allow execution of the total system in debug mode.

4.3.4.7 The execution software shall include the gate level emulation algorithm.

4.3.4.7.1 The gate level emulation algorithm shall be table driven, using the gate tables produced by the translator specified in paragraph 4.3.2.2.3.

4.3.4.7.2 The gate level emulation algorithm shall be able to emulate at least a 6000 gate system.

4.3.4.7.2.1 The maximum slow down factor for the algorithm operating on a 6000 gate system, assuming an average gate fan out of 2, 5% of gates changing value in any one emulated machine cycle, shall be 3000 times slower than real

time. This timing should be based on a 1 microsecond emulated machine cycle time.

4.3.4.7.2.1.1 This requirement is based on the results of the feasibility study (Attachment 1).

4.3.5 Analysis Software

Software to support the reduction of the data gathered during test execution shall be provided. This software shall provide for data reduction, statistical analyses and reliability modeling.

4.3.5.1 The data reduction software shall allow the analyst to group common data and reduce it to necessary components, using the data recorded during the test execution phase. It shall allow the extraction of items deemed important for a particular use on an individual basis by the analyst.

4.3.5.2 The statistical analysis software shall provide the capability to calculate statistical parameters from the reduced data produced above. The following capabilities shall be provided at a minimum:

1) Matrix manipulation

   Real Matrices

   Complete Matrices

   Eigen values, Eigen vectors

2) Ordinary differential equations

3) Regression analysis

4) Time series analysis

5) Variance analysis

6) Interpolation

7) Numerical integration

8) Differentiations

9) Polynomial manipulations

4.3.5.3 The reliability modeling software shall provide the capability to develop parameterized, unified reliability models of the system of interest.

4.3.5.3.1 The reliability model shall be capable of using the statistical data produced from actual test execution as an input in place of predicted or expected paramters.

4.3.5.3.2 The reliability model software shall suppport development of models for fault tolerant, multiple processor systems.

APPENDIX A

Hardware Composition Trade Study

Table of Contents

I.	Introduction	A-1
II.	Methodology	A-1
III.	Evaluation	A-2
IV.	Final Recommendation	A-2

I. Introduction

This trade study was done to determine the best approach to the hosting of the various pieces of software needed by the digitial avionics design and reliability analyzer. The trade study was designed to answer the question: "should the facility be based on an emulator-only system or emulator/support machine system?". The emulator/support system envisions an emulation machine connected to general purpose computer. The general purpose computer supports most of the software, with the emulator supporting only actual emulations. In the emulator-only system, the emulator must support everything.

II. Methodology

To perform the trade study the following 5 criteria were established:

1) Operation speed
2) User interface
3) Difficulty of use
4) Cost of implementation
5) Size of facility needed

These criteria were then rank ordered in order of importance (as shown in the list above) and assigned weights of 5 to 1 with 5 being the most important (operation speed).

Each alternative was then evaluated for its satisfaction of each criteria on a scale of 1 to 10 with 10 being most satisfactory and 1 the least. We then multiplied the satisfaction by the criterion weighting to obtain the weighted ranking. Weighted rankings for each criterion were then added to give a total for each alternative, with the higher score reflecting the "best" choice. The results are shown in Table A-1.

Table A-1 Hardware Composition Trade Study

	Operation Speed (5)	User Interface (4)	Difficulty Of Use (3)	Cost Of Implementation (2)	Size Of Facility (1)	Total
Emulator Only	3 / 15	4 / 16	5 / 15	5 / 10	4 / 4	60
Emulator/Support	6 / 30	6 / 24	6 / 18	3 / 6	4 / 4	82

III. Evaluation

1) Operation Speed - The feasibility study indicated that the primary limiting factor for the avionics design and reliability analyzer in the gate level emulation mode is the speed of the gate level emulation. In the emulator/support case, the support computer removes the burden for support of environmental simulation etc., from the emulator. Thus this combination rates 6. A more parallel system could get a higher score. The emulator only system rates a 3.

2) User Interface - The user interface is one of the key items in the use of the facility. If this interface is poor, there will be a reluctance to use the system. General purpose support machines have the user interface as one of their most visible portions and hence, modern operating systems have attempted to provide for a flexible interface. Emulators, on the other hand have a much narrower applicability and hence less attention is paid to such "mundane" factors.

3) Difficulty of Use - This relates not only to the user interface, but also to the operating system backing it up. The emulator/support combination can use the general purpose operating system to hide the tedious details of interaction with the emulation machine which is attached to it. In the emulator-only case, the user usually has to explicitly deal with the details of the emulation machine.

4) Cost of Implementation - The emulator/support system represents an increase in hardware cost over the emulator-only system. Comparable software needs to be developed in both cases, with the exception of the additional driver software necessitated by the emulator/support interface.

5) Size of Facility Needed - There is no clear indicator that either choice represents a better possibility here.

IV. Final Recommendation

Based on the established criteria, the emulator/support system is recommended.

APPENDIX B

Microprogrammable Computer Trade Study

Table of Contents

I.	Introduction	B-1
II.	Microprogrammable Computer Architecture	B-1
III.	Requirements	B-2
IV.	Computer Search	B-3
V.	Computer Performance Analysis	B-6
VI.	Conclusion	B-15

List of Tables

Table	Page
B-1	B-4
B-2	B-5

I. Introduction

The following trade study was done to determine which microprogrammable computers would best serve as the emulator portion of the digital avionics design and reliability analyzer. First a search was done to find all available user-microprogrammable machines. These were then analyzed to determine which ones met the requirements for implementing the NASA Langley gate-level emulation algorithm. The machines which met the requirements were then compared concerning performance and price. A select few were recommended as candidates for the emulator portion of the digital avionics design and reliability analyzer.

II. Microprogrammable Computer Architecture

A microprogrammable computer is one whose microcode can be changed by the user. Microcode, which is stored in control store, consists of microinstructions which control the primitive operations of the computer. A complex operation performed by a computer can be represented as a sequence of microoperations. There are three types of microinstructions: vertical, allowing one operation per instruction; diagonal, allowing one or more; and horizontal, allowing many operations per instruction, thus increasing processing speed.

III. Requirements

There are a number of requirements that must be met by a user-microprogrammable computer in order to implement the NASA Langley gate-level algorithm. These requirements are based on a feasibility study implementation of the algorithm using the Nanodata QM/1 computer.

1. The microcode controlling the machine shall be user-programmable through software.
2. The microcode shall provide for parallel operations within a single microword.
3. Control shall be directed from main store via a gate info word of at least 8 bits to micro store using a vector mechanism. The gate info word contains the address of the location in micro store to which control is directed.
4. Control store shall contain at least one thousand words.
5. Main memory shall be sufficient to handle the algorithm, at least 32 thousand words.

IV. Computer Search

Various references were investigated in order to find names of all companies manufacturing minicomputers which are microprogrammed. The following sources were used: Auerbach Publishers, Inc., Data Pro Information Services, Electronic Buyers' Guide 1980, NASA Recon Data Base (remote console), Defense Technical Information Center, and the Lockheed DIALOG data base. This search resulted in the list of companies shown in Table B-1. Each company was then contacted and asked which, if any, of their minicomputers were user microprogrammable and could function as emulators. The list of computers shown in Table B-2 resulted from these inquiries.

Table B-1. Computer Manufacturers Surveyed

MANUFACTURER	COMMENT
1) Burroughs	See Table B-2
2) Cado Systems	Word length limited to 8 bits
3) Control Data Corp.	See Table B-2
4) Data General	See Table B-2
5) Digital Equipment Corp.	See Table B-2
6) Digital Scientific	See Table B-2
7) Hewlett Packard	See Table B-2
8) Honeywell	See Table B-2
9) Nanodata Corp.	See Table B-2
10) Northrop Data Sys.	Nothing user microprogrammable
11) Microdata	Nothing user microprogrammable
12) Ohio Scientific	Nothing user microprogrammable
13) Perkin-Elmer	See Table B-2
14) Prime Computer Inc.	Only limited information available
15) Rolm	Machine too small; C.S. too small
16) Sperry Univac	See Table B-2
17) Systems Engineering Lab	See Table B-2

TABLE B-2. MICROPROGRAMMABLE COMPUTER CHARACTERISTICS

MACHINE	MAIN STORE		CONTROL STORE			PRELIMINARY CANDIDATE	PERFORMANCE RATING	COST		PERFORMANCE PER COST RATING	FINAL CANDIDATE
	WORD LENGTH (BITS)	CYCLE TIME (NS)	MAX. WORD CAPACITY	WORD LENGTH (BITS)	CYCLE TIME (NS)						
Burroughs 1900	16	500 82	4K (cache)	16 (micro)	82	Yes	7	$71,500 12,000 83,500	CPU, 128K Cache	8	No
Control Data Corp. Cyber 18	16	600 or 900	8K	32 (micro)	168	Yes	7	$19,300 4,266 23,566	CPU,32K C.S	30	No
Data General Eclipse S250	16	500	1K WCS 2K PROM	56 (micro)	200	Yes	12	$36,000 4,200 40,200	CPU, 732K WCS	30	Yes
Digital Equipment VAX 11/750	32	800	(6K C.S.) 1K WCS	80 (micro)	70	Yes	14	$47,000 10,700 57,700	CPU, 732K WCS	24	Yes
Digital Equipment VAX 11/780	32	900	(4K C.S.) 1KWCS	96 (micro)	70	Yes	14	$128,000 10,700 138,700	CPU, 256K WCS	10	No
Digital Scientific Meta 4 5000	32	350	2K	32 (micro)	135	No	N/A	N/A	N/A	N/A	No
Hewlett Packard 1000 F	16	350	15K	24 (micro	250	Yes	7.5	$11,750 2,000 13,750	CPU,32K C.S	54	No
Honeywell Level 6 Model 43	16	300	2K	64 (micro)	280	Yes	13.5	$10,200 9,240 19,443	CPU,32K C.S	69	Yes
Nanodata QM/1	8	750	1K 32K	144 (nano) 18 (micro)	80 150	Yes	59	$176,000	CPU, 40K WCS	33	Yes
Perkin-Elmer 3240	32	500 (340 ns)	2K	32 (micro)	50	No	N/A	N/A	N/A	N/A	No
Sperry-Univac V77-600	16	660	4K	64 (micro)	190	Yes	11.5	$16,800 7,500 24,300	CPU,32K C.S	47	Yes
Sperry Univac V77-800	16	600	1K	48 (micro)	150	Yes	11	$33,000 4,000 37,000	CPU, 128K WCS	30	No
Systems Engineering Lab 32 Series	32	600	4K	64 (micro)	150	Yes	13	$16,000 8,000 24,000	CPU, 732K WCS	54	Yes

V. Computer Performance Analyses

Each computer was then examined to determine whether or not it would meet the requirements determined by the study done on the QM/1 using the NASA Langley Research Center gate-level emulation algorithm. All machines met both the 32K main store requirement as well as the 1K control store requirement. The following analyses discuss the operation of each machine in relation to Requirements 1, 2, and 3.

1.0 Burroughs B1800 or B1900 Series

In this computer cache memory (2K words) is used as control store; it is possible to store all of the microcode in the cache memory. A pipelined processor permits fetching, decoding, and executing microinstructions to be performed separately and concurrently thus compensating for the limited capability of the 16-bit microcode. Memory addressing at the hardware and microcode level is accomplished through a 24-bit field address register that can directly address 16,777,215 bits as though they were a continuous string. Up to 24 bits can be processed in one operation taking 167 ns. Optional port interchange enables independent rather than processor-dependent access to main store by such devices as the multi-line data communications control. The 18-bit A register contains the absolute "S" or Main Memory address of the microinstruction to be executed.

This machine would be a suitable candidate.

2.0 Control Data Cyber 18

The CDC Cyber 18 was designed to emulate the CDC 1700 Series. The microprocessor contains 2K to 4K of 32-bit user programmable microcode. One type of micro memory consists of 512 words of read/write memory and/or 1K words of read only memory; the other type contains 2K of read/write memory. Each 32-bit microinstruction is divided into five main sections each performing a different operation in parallel with the others. The microprocessor controls the machine at all times. The process of decoding a macroword in main store determines the address of the micro routine which is called.

The read/write random access memory (RAM) can either be loaded from an external device or data can be written into micro memory under control of the micro program.

Since this machine does have sufficient control store of parallel microcode and uses a vector mechanism to transfer control from main store to micro store it would be a candidate.

3.0 Data General Eclipse

The control store of the Data General Eclipse contains 2K 56-bit words of parallel microcode. Each microinstruction is divided into 15 micro fields which can be grouped according to the purpose they serve. A word in control store is addressed by the 12-bit output of the state change logic which is determined by the contents of the True Address bus or the False address field. In order to start main memory the CPU places an address on the logical address (LA) bus and issues a start signal to memory. Only the module containing the memory location addressed responds to the signal.

There is a microassembler available to enable the user to write microprograms in symbolic form and assemble them to produce a binary object file. The microloader is then used to load the object files.

This machine contains the vector mechanism to address the microcode and has flexible, parallel microcode so it would be a candidate.

4.0 DEC - VAX 11/750

The VAX 11/750 contains 6K of 80-bit microcode. A single microinstruction can perform many operations in parallel. The VAX 11/750 was designed as an emulator for the VAX architecture and contains 1K of user control store. Emulation starts with one micro-order called the BUT/IRD1. This signals the beginning of the next VAX machine instruction. In the micro-code which emulates each VAX instruction, this micro-order is present in the last microinstruction. Access to the user control store is by the opcode called "FC" in the VAX instruction stream. This opcode results in a branch to a location in user control store. From this point on, user microcode has control of the micromachine. Control can then be returned to the VAX emulation by means of the BUT/IRD1 micro-order.

There are a number of features which support user microprogramming: the data path which includes 18 general purpose 32-bit scratch pad registers, 8 of which have ports to both the RBUS and the MBUS; the super rotator, which allows very efficient (in hardware) bit picking operations; and a flexible ALU. The microsequencer supports general microprogramming in three important ways; conditional branching, loop control, and subroutine control. The VAX 11/750 has six independent flag bits, four of which are always available for user microprogramming and two of which are conditionally available. There is a 5-bit step counter which can be initialized to any arbitrary value ($0 \leq X \leq 30$). For subroutine control, a 16-deep microstack is available for nested subroutine calls.

This computer does have the required horizontal microcode as well as an opcode resulting in a branch to user control store so would be a good candidate.

5.0 DEC - VAX 11/780

The VAX 11/780 contains 1K of 96-bit user control store which is available primarily for augmenting the speed and power of the basic machine. It is, however, possible to access 4K of ROM containing the operation and sequencing of the central processing unit. The architecture and operation of the VAX 11/780 is similar to the VAX 11/750 as far as the requirements of this contract are concerned.

This machine would be a good candidate.

6.0 Digital Scientific META 4

This machine is designed to be an adjunct processor to a main CPU. One possible application is as an I/O processor. The microcode instruction set is very structured, 32 bits long. Typical predefined instructions include load from control store, move register to register, etc. Control store size is limited. Microcode can also read from "main store" via a request, wait protocol.

Microcode operation is not started via a vectored operation and in general, this "microprogrammable" machine is typical of a mini computer without microprogrammability.

This computer will not provide the capabilities necessary for our purposes. Microcode execution is not started via an opcode type operation, necessitating bit decoding in the implementation of the algorithm. The machine does not have ready access to the larger main store which would be necessary to hold gate tables in the algorithm. Finally, the instruction set looks like a mini computer instruction set and is not flexible enough to do the bit manipulation we need.

7.0 Hewlett Packard 1000 E/F Series

The HP 1000 E/F Series has 50K of user addressable 24-bit microcode in control store with access to 12 scratch pad registers. There are four word types of microcode with up to five micro-orders each. Each micro-order defines one or more operations to be performed by the computer.

The control processor, part of the CPU, is always in control of the computer, and the base set microroutines cause the read operations to occur for all instructions and data from main memory. All 16-bit instructions are placed in the Instruction Register (IR) and decoded. The process of decoding the IR bits determines which control memory address (which microprogram) is called by the instruction received from main memory. Control memory module selection is determined by the value of bits 8 through 4 in the Instruction Register. These bits help determine the address of branches in the control memory base set Primary Mapping Table, which in turn directs a branch to the desired module.

There is a micro programming support software package consisting of the following:

- RTE Microassembler Program
- RTE Microassembler Cross-Reference Generator Program
- RTE Microdebug Editor Program
- RTE Microdebug Editor Subroutine
- RTE Driver DVR36
- WCS I/O Utility Routine WLOAD
- PROM Tape Generator Program

The microcode may be loaded into writable control store (WCS) modules or may be permanently fused in programmable read-only memory (PROM) chips.

This machine contains horizontal microcode as well as the necessary vector mechanism so would be a good candidate.

8.0 Honeywell Level 6

The Honeywell Level 6 contains up to 2K 64-bit words in its writable control store. Each 64-bit word is divided into four 16-bit segments each of which can be loaded with a separate instruction. Thus, one word may perform four parallel operations. Control is transferred from the CPU to the writable control store by causing the CPU to issue a megabus cycle (I/O write) addressed to the WCS. This operation is performed by the native firmware whenever the first word of an instruction lies in the range 0080 hexadecimal through 00BF hexadecimal (64 bits). The location to which control is transferred is one of the first 16 locations in the WCS; the specific location is identified by the least significant hexadecimal digit of the instruction word.

There is a WCS assembler available to assemble firmware routines as well as a loader to load the assembled routines into the WCS. A microcode analyzer is available to selectively display pertinent CPU and WCS information for debugging microprograms.

Due to the horizontal microcode and the vectoring effect transferring control from the CPU to the microcode, this machine would be a candidate.

9.0 Nanodata QM/1

The Nanodata QM/1 is unique in that it is specifically designed to emulate other computers. There are two levels of microprogramming with the lower level called nanoprogramming. The top level microprogram is an 18-bit vertical microcode having many of the characteristics of an assembly language. The lowest level microcode is a 360-bit horizontal word (144 bits of which are active at any one time) which interprets the higher level microcode. The identification of the nanoword which interprets a given microinstruction is determined by 7 bits in the 18-bit microword itself and a 3 bit page indicator in a CPU store register, giving a total of 10 bits of address to cover the 1024 words of nanostore.

The control store limit is 40K words. For the Langley algorithm, the algorithm would be coded in nanocode, using control store to provide the vector into the proper nanoword and to hold the gate state information. Based on its architecture and the actual implementation of the Langley algorithm for the QM/1 under the feasibility study, the QM/1 is a suitable candidate.

10.0 Perkin Elmer 3320

The Perkin Elmer 3320 contains 2K 32-bit words of writable control store. WCS is addressable through ROM location counter (RLC). There are four assembly level instructions which enable the user to write into WCS, read from WCS, and transfer control to WCS resident microcode. Unfortunately the 2K words of the WCS serve as a supplement to the fixed control store; the user cannot delete or modify user level instructions or machine features located in the ROM control store. If an operation does not exist in ROM, it cannot be used in WCS. A new emulator cannot be created in WCS; the user can only add to the existing one. For this reason, this machine would not be a suitable candidate.

11.0 Sperry Univac V77-800

The Sperry Univac contains 2K of 48 bit microcode in writable control store (WCS) with space for 1K 48-bit ROM storage. Each microword executes multiple operations. The WCS acts as an extension of the processor control store.

The WCS contains a decoder control store, a central control store (CCS), and an I/O control store. The decoder control store consists of two 16-word by 16-bit memory arrays with associated logic that decodes main memory instructions into a 9-bit address which is applied to the CCS. Addressing for the 64-bit microinstruction is provided by the 9-bit address from either the processor, decoder control store, or subroutine stack.

The microcode is input as a series of source statements via a terminal or card reader using the operating system VORTEX II or SUMMIT. The Microassembler, MIDAS, is then used to transform these statements to object code. The object code is then loaded into WCS using the microutility, MIUTIL.

This machine does contain horizontal microcode as well as the necessary vector mechanism to control store and would be a viable candidate.

The architecture of the V77-600 is the same except that there are 4K 64 bits of WCS. Thus this machine would also be a candidate.

12.0 Systems Engineering Laboratories 32/70 Series

The SEL 32 Series contains 4K 64 bit high speed Random Access Memory (RAM) as a physical extension of Control Store (CROM). The microinstructions contained in WCS allow parallel operations within the execution timing of a single instruction.

The writable control store (WCS) may be used as a CROM extension in the host computer, or it may be used with the Development Support System (DSS), residing in the DSS Test Stand. The CROM takes an instruction from Main Memory and stores it in a 32-bit internal register (I1). An appropriate microprogram is executed and the contents of register I1 are moved to register IO (a 32-bit register). The CROM entry point is determined by a decode of the contents of register IO. The CROM contains a series of read only memories (ROMs) which contain the decode and vector tables within CROM to the microprogrammed routines that operate the computer.

Entry into the WCS from software is accomplished using the JUMP WCS Macro-Assembler instruction. This instruction allows the user to jump to any of the first 64 locations in WCS where vector addresses (in microcode) are stored, which address routines within the WCS.

The writing of WCS is accomplished using the WRITE WCS Macro-Instruction. The reading of WCS is accomplished using the READ WCS Macro-Instruction.

Since this machine does have horizontal microcode and does have the vector mechanism from main memory to control store it would be a suitable candidate.

In addition to the analyses that were done to determine whether each computer met the requirements, an algorithm was used to rank the computers with respect to those characteristics necessary to the solution of the gate-level algorithm[1]. The following equation was examined then altered to better fit the algorithm requirements:

$$P = \frac{10^{12}\left[(L-7)\ (T)\ (WF)\right]^{i}}{\left[32{,}000\ (36-7)\right]^{i}\left[tc - t_{I/O}\right]}$$

where

P= the computing power in bits per second

L= the word length in bits

T= the total number of words in memory

WF= 1 for fixed word length memory

2 for variable word length memory

tc= the time in microseconds for the CPU to perform one million operations

$t_{I/O}$= the time the CPU sits idle waiting for I/O to take place

i= .5

---

1 Knight, Kenneth E.: Changes in Computer Performance, Datamation, vol. 12, no. 9, pp. 40-54, September 1966.

The above equation was altered to include only those parameters relevant to the implementation of the NASA LRC gate-level emulation:

$$P' = \frac{10^{12} \left[(L-7)\ (1)\ (1)\right]^{1/2}}{\left[(32{,}000)\ (36-7)\right]^{1/2}} \Bigg/ \frac{\left[2\ (CS) + (m)\right]}{3}$$

where

CS= control store cycle time in microseconds

M= main memory cycle time in microseconds

P= a measure of the bits processed based on a weighted average cycle time

A weighted average of the control store cycle time and the memory cycle time was chosen as the control store is accessed more frequently than the main memory so its access time should carry more weight in analyses of the overall performance.

The measurement P' is not meant to be a direct measurement of the power of each machine but more of a relative measurement of performance to aid in choosing the computer which best fills the requirements of this contract.

The value for P' for each computer was then scaled to fall between 1 and 100 in order to more easily rank the performances. These numbers appear in Table A-2 under the heading "Performance Rating". The value for the performance rating was then divided by the cost of the CPU with minimum memory (at least 32K words) plus control store to give a value for performance per dollar. These values were then scaled to fall between 1 and 100 to give each candidate a "Performance per Cost Rating".

The prices quoted in Table B-2 represent only the price of the CPU with a minimum of memory plus the control store. They do not reflect the price of interfaces, consoles, printers, etc. They should be used only as a general basis for cost comparison.

VI. Conclusion

The machines which have been recommended as final candidates were chosen more on a basis of performance than cost due to the stringent requirements for supporting the gate-level algorithm.

The performance of the QM/1 is far superior to any of the other machines studied. There are a number of machines that compete for second place such as the DEC VAX 11/750, DEC VAX 11/780, Honeywell Level 6 Model 43, Systems Engineering Lab 32 Series, Data General Eclrose, Sperry Univac V77-600, and Sperry Univac V77-800. Comparing the two VAX machines, one would eliminate the VAX 11/780 on basis of cost. The Univac V77-800 could be eliminated for the same reason. Any of the following machines would be good second choices:

1) DEC VAX 11/750
2) Honeywell Level 6 Model 43
3) SEL 32 Series
4) Univac V77-600
5) Data General

The QM/1 far outperforms those machines in second place and would be the recommended choice for the emulator portion of the Digital Avonics Design and Reliability Analyzer.

Attachment 1

Interim Technical Report

Table of Contents

I.	INTRODUCTION	1
II.	SUMMARY	3
III.	BASIC ALGORITHM DESCRIPTION WITH PRELIMINARY TIMING ESTIMATE	5
IV.	ADDITIONAL FEATURES OF THE ALGORITHM	24
V.	IMPLEMENTATION OF THE ALGORITHM	29
VI.	TIMING RESULTS	37
VII.	CONCLUSIONS	49
VIII.	REFERENCES	51
Appendix A.	UNIFORMITY OF GATE TREATMENT	A-1 thru A-6
Appendix B.	DERIVATION OF EQUATIONS	B-1 thru B-11
Appendix C.	NANOCODE FOR BEST CASE TIMING ESTIMATE	C-1 thru C-7

I. INTRODUCTION

This interim technical report details the results of Martin Marietta's implementation on the Nanodata QM/1 of an algorithm for the emulation of digital devices at the gate level. The implementation is intended to prove the feasibility of using emulation technology for data collection in support of reliability studies of fault tolerant digital avionics equipment. From the high level point of view, it is clear that that feasibility depends primarily on the adequacy of the speed improvements emulation seems to offer over simulation. That is to say, the most useful measure of the feasibility is the time required to perform a "sufficient" number of experimental runs to give statistical significance to the results obtained.

The specific algorithm which we implemented was developed by the NASA Langley Research Center. The algorithm has two significant factors inherent in its use. First, it doesn't require examination of every gate in the system and second, it allows treatment of every gate in the same manner, regardless of gate type. The algorithm is described in detail in sections III, IV, V and Appendix A.

To provide a basis for the actual timing figures, Section III provides a discussion of the basic operations involved in the algorithm and the basic timing considerations in the QM/1 to try to determine a "best case" slow-down factor for the algorithm (i.e., an indication of the best we can do in terms of speed). This section contains a brief, high level overview of the philosophy of the algorithm and provides an introduction to the more complex discussion in Section IV. Section V gives details of our implementation including considerations

of memory requirements and system size. The timing results are detailed in Section VI and include graphs to determine predicted performance for systems of varying sizes. Section VII presents our conclusions based on the implementation and timing studies. Appendix A contains the rationale and basis for uniformity of gate treatment, Appendix B contains the derivation of the timing equations used for projection, and Appendix C contains the nanocode for the "best case" timing analysis.

II. SUMMARY

This report details the results and conclusions of Martin Marietta's implementation of the NASA Langley Research Center's gate level emulation algorithm on the Nanodata QM/1 computer. The implementation was done to determine the applicability of emulation technology to reliability analysis of digital avionics systems. This determination has focused primarily on the speed aspects of the emulation and the time necessary to run a sufficient number of sample cases to provide significant results.

The slow-down factor of the emulation is based on four primary considerations:

1. The system size (number of gates);
2. The average percentage of gates changing value in a machine cycle;
3. The average fan-out of the gates in the system;
4. The machine cycle time of the system under study.

Using a system size of 2000 gates, and assuming 5% of the gates change value, with an average gate fan-out of 2.0, and a machine cycle time of 1μs, the actual slow down factor based on the implementation was found to be 1200:1. This is compared to a best possible slow-down of 600:1. The 1200:1 figure means that 10,000 samples of 0.1 seconds real time per sample would take 17.2 days of emulation processing, a span which is entirely reasonable in relative to the kinds of numbers seen in previous studies (i.e.[3]).

The largest control-store resident system possible under the same constraints (6000 gates) also exhibits a reasonable slow-down factor

of 3500:1. However, in attempting to extend the emulation capability beyond 6000 gates, we found that the processing time is overshadowed by the time it takes to load data into control-store, and hence this mode of operation is not feasible.

The basic conclusion of the report is that gate level emulations of systems up to 6000 gates is feasible within the constraints imposed by the architecture of the QM/1.

III. BASIC ALGORITHM DESCRIPTION WITH PRELIMINARY TIMING ESTIMATE

In our discussions with NASA Langley Research Center Personnel, we have been given several estimates of slow-down factor expected by them in the QM/1 implementation of their algorithm. These have ranged from a low of 300:1 to a higher range of 500-600:1. From our implementation of the algorithm, these figures seemed very optimistic. The following discussion is an attempt to define a possible, reasonable lower bound on the slow-down factor, taking into account QM/1 and nanocode realities as well as the operations necessary because of the algorithm.

For the analysis which follows, we assume that the reader is familiar with the basics of the QM/1 and its nanocode. We further assume that the reader is familiar with the algorithm implemented (described briefly below). Please note that in these assumptions we do not require a working knowledge of either item, only a familiarity to the extent that allows an understanding of the terms involved. For example, most of our discussion will be based on the basic time cycle in the QM/1, the T period (80 ns). It is sufficient for the reader to realize what a T-period is and what it means in relation to execution of a nanoword.

The NASA LRC algorithm is conceptually straightforward. There is one item requiring acceptance on the reader's part. The algorithm as defined allows all gates of any type (AND, OR, NAND, NOR, XOR, etc.) to be treated identically after initialization of the value of the gate and a quantity called CNT (count) which relates to the number of inputs. The algorithm has a major and a minor loop as shown in Figure III-1.

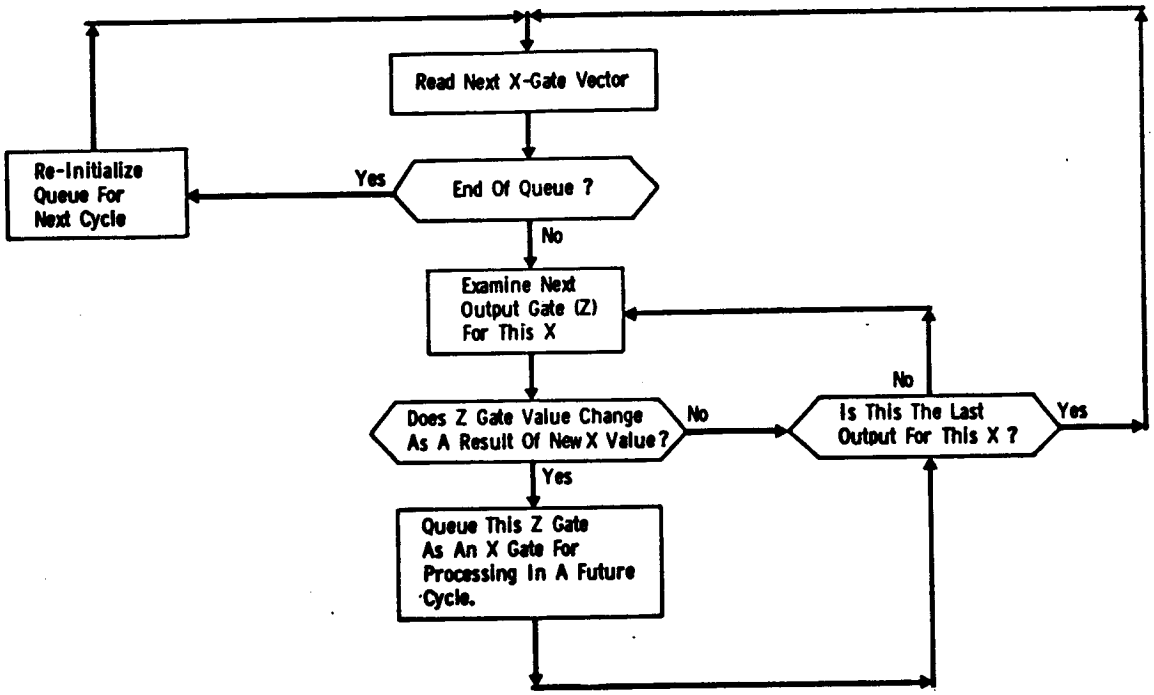


Figure III-1 ALGORITHM OVERVIEW

The driver on the major loop is a queue consisting of those gates whose value has changed. The minor loop is comprised of examining each gate which is connected to the output of the changed gate to see if the change affects the value of the output gate. For example, we will use the 3 gates shown in Figure III-2. Suppose the value of gate A has changed (the mechanism for this is described later). The algorithm

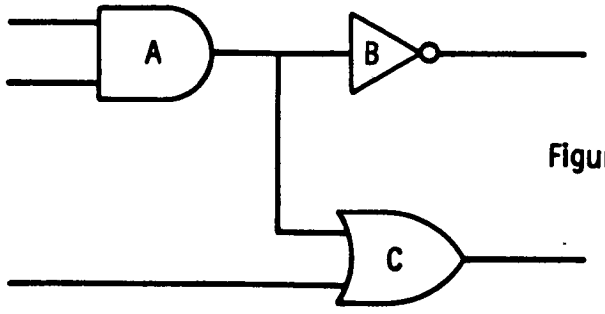


Figure III-2

then specifies that gates B and C must be examined to determine if the change in the value of A will cause a change in the value of either B or C. This is determined by updating the CNT quantity for the output gate (B or C) based on the new value of the input (A) and determining if CNT, by virtue of that update, transitions into or out of zero (see Appendix A for an explanation of this mechanism). Transition of CNT for the output gate into or out of zero indicates that the value of the output gate (B or C) changes value. If the value of the output gate (B or C) does not change because of the input (A) value change, no further action is necessary. If the value of the output gate (B or C) does change, the gate is added to the next cycle's changed-value queue (as a future x-gate) to allow examination of the effects on its outputs.

For brevity in our discussion, we will term the changed-value queue and its processing "the x queue" and "x processing". We will similarly term the output gate processing to be "z processing". The action of the algorithm then causes a z whose value changes to become

an x for the next "cycle" (for its outputs to be examined). In this case, a "cycle" represents the propagation of the signal through one logic level; and a "machine cycle" would be completed when the x queue becomes empty (i.e., when the logic circuit has reacted to changed inputs and the circuit has settled to quiescent values).

There are several benefits derived from the algorithm. For x processing, only those gates whose values change need to be examined. Further, only the outputs of those gates must be known. This contrasts with the typical "brute force" algorithm which requires examination of each gate and subsequent examination of all of its inputs to determine its value.

For the analysis below, the only way a gate is put on the x queue is by virtue of its having been examined during z processing and found to have changed value. We will term this the non-null case of z processing. Thus we have the following relationship

$$\#\ x\ \text{gates processed} = \#\ \text{non-null z gates processed} \qquad (1)$$

This ignores the mechanism for starting the cycle, so we will cover that later. The null z process is the other case for z's and represents basically no operation (i.e., no action necessary since the gate doesn't change value).

To provide some quantitative values for our discussion we need to make some assumptions concerning the system being modeled at the gate level. For example, we need to fix the size of the system. This is due to the fact that the slow-down factor is directly proportional to the system size (actually the proportionality is based on the number of gates whose

values change, but this is related to system size). Gate processing must proceed sequentially, while the modeled machine cycle time is fixed. Therefore, we based our analysis on the following assumptions:

1. The system under consideration contains 2000 gates;
2. Only 5% of the gates will change value in any machine cycle (x processing);
3. The average gate fan-out is 2 (there are 2 z's per each x);
4. The basic machine cycle time of the emulated system is 0.1μs.

For discussion purposes, any further reference to cycle, cycle time or machine cycle will refer to the modeled machine cycle time and represents the time for the logic circuit in the modeled machine to react completely to a change in input. This represents the real time against which the algorithm is measured.

A second set of assumptions is necessary for this analysis. This second set relates to the data structure upon which the algorithm is built. For this discussion we will assume the following structure (shown pictorially in Figure III-3).

1. Each gate is characterized by a gate info word. This word contains the gate value, CNT and various other information.
2. The x queue is a linked list with the link word following the gate info word in memory. The link word contains the address of the next gate info word in the queue. We will call the link word "LINK".

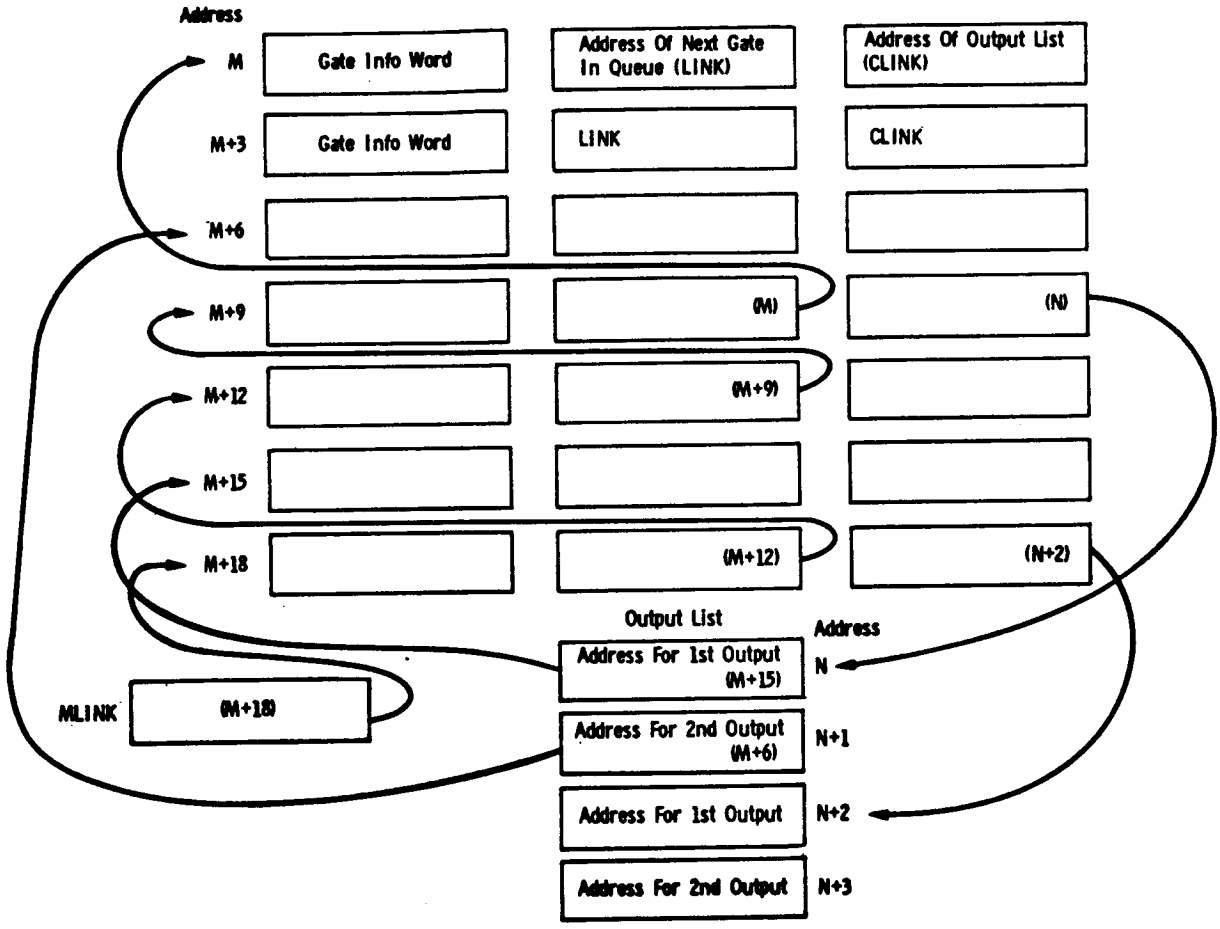


Figure III - 3 Gate Queueing Structure

3. The address of the first gate info word in the queue is kept in a local store register designated MLINK.
4. The output gate addresses are kept in a separate section of control store in consecutive order. That is, the address of the gate info word for output 2 of gate y follows immediately the address of the gate info word for output 1 of gate y.
5. The address of the output list for each gate is contained in a word following the queue link word (LINK). We will call this word "CLINK."

From the diagram of Figure III-3, we see the gate whose info word is located at address m+9 has two outputs. The first output is the gate whose info word is located at address m+15 and the second is the gate whose info word is located at address m+6. This is found by following the CLINK to address n to address m+15 and then following address n+1 to address m+6. The x queue in that figure goes from the gate of address m+18 (due to MLINK) to m+12 (LINK) to m+9 (LINK) to m (LINK).

The data structure just presented represents, in a somewhat simplified manner, the data structure used in our actual implementation. The specifics of the implemented algorithm are covered in more detail in Section V.

Given these assumptions concerning system size, the definition of the algorithm and the structure of the data in QM/1 control store, we can now begin to analyze the potential slow-down factors based on those assumptions. One further assumption which is inherent in the following analysis is that the decisions concerning

the actions to be taken in z processing are made in a highly parallel fashion using the QM/1 microinstruction execution feature [1:66] and local store register R31. Using this feature, testing of bits in the gate info word is done quickly in a highly parallel fashion requiring a very small amount of time.

For the first cut estimate of slow-down, we will ignore the time necessary for x processing and concentrate on what is required for z processing. However, to determine the number of gates examined in z processing, we must use system assumptions 1 and 2 to determine the number of x's processed and then multiply that by the 2 from system assumption 3 to give us the total number of output gates examined (i.e., 2 outputs per x = total z's). Therefore we have the following:

$$\begin{aligned} \#\text{ x gates processed} &= (\text{system size}) \text{ X } (\% \text{ system changing}) \\ &= 2000 \text{ gates X } 5\% \\ &= 100 \text{ gates} \end{aligned} \tag{2}$$

$$\begin{aligned} \#\text{ z gates processed} &= \#\text{ x gates processed X fan-out of x gates} \\ &= 100 \text{ gates X } 2 \\ &= 200 \text{ gates} \end{aligned} \tag{3}$$

Now using equation (1), we find the number of non-null z cases. Equation (1) stated

$$\#\text{ x gates processed} = \#\text{ non-null z gates processed} \tag{1}$$

Therefore, we have

$$\begin{aligned} \#\text{ non-null z gates processed} &= \#\text{ x gates processed} \\ &= 100 \text{ gates} \end{aligned} \tag{4}$$

Which means that of the 200 z gates we have, half are the non-null case and the other half are the null case. Now that we have fixed the amount of processing to be done, we need to get an estimate of the time necessary to do each case. We will consider the null case first. The shortest nanoword in the QM/1 which does not branch to itself looks like [1:58]:

```
Tn:      READ NS (not stretched)
Tn+1:    GATE NS (not stretched)

                 or

Tn:      STRETCH, READ NS, GATE NS
```

We need to explain our notation in the above two examples. The Tx to the left indicates the T-step (not T-period which is fixed, but T-step which may be either 1 or 2 T-periods long). In the first case, the T-steps are not stretched which means they are each 1 T-period long. In the second, the T-step is stretched indicating it is 2 T-periods long. For those more familiar with nanocode notation, this can be shown thus:

```
X . . .   READ NS
.X . .    GATE NS

                 or

S . . .   READ NS, GATE NS.
```

The net result is that the null z processing requires, in the best possible case, 2 T-periods.

For the non-null case, we need to do more in the nanoword than simply branch out. Let us assume that we can do all necessary processing in one nanoword. To determine the length of that nanoword let us examine the length of a set of nanowords. The set of MULTI nanowords, consisting of 124 words, represent 669 T-periods. This works out to:

$$\begin{aligned} \text{T-periods/nanoword} &= 669 \text{ T-periods}/124 \text{ nanowords} \\ &= 5.39 \text{ T-periods/nanoword} \end{aligned} \tag{5}$$

This result fits in well with intuition in which we realize that the case where none of the T-steps in a nanoword are stretched is relatively rare, and that in most words observed, at least one and occasionally 2 of the T-steps are stretched. Thus, without considering the exact operations to be performed, we will use a 5.4 T-periods/nanoword figure for the non-null z processing nanoword.

Now we have the information necessary to calculate the z processing and the absolute best case slow down factor. The time required is given by

$$\begin{aligned} \text{null z processing} &= \#\text{ null z gates X 2 T/gate} \\ &= 200 \text{ X } 2T \\ &= 200T; \end{aligned}$$

$$\begin{aligned} \text{non-null z processing} &= \#\text{ non-null z gates X 5.4 T/gate} \\ &= 100 \text{ X } 5.4T \\ &= 540T; \end{aligned}$$

total z processing = null z processing + non-null z processing

= 200T + 540T

= 740T.

To translate this to understandable terms, we use the 80ns/T-period conversion to get:

time = 740T X 80ns/T

= 59200ns = 59.2μs

For our given machine cycle time of 0.1μs (system assumption 4), the slow-down factor is given by:

$$\text{slow-down factor} = \frac{\text{actual time}}{\text{machine time}}$$

$$= \frac{59.2\mu s}{0.1\mu s}$$

= 592:1

This 600:1 factor is close to the NASA LRC expected slow-down in their 500-600:1 estimate. What is significant about this figure is that our judgment about 600:1, which appeared to be an optimistic figure is proven to be true. This slow-down factor is based solely on z processing, does not include x processing at all, and in addition, does not include most of the processing necessary for z's.

Let us look further into the x processing. This processing must consist minimally of:

1) Reading in the x gate info word from control store;
2) Reading in the address of the output list (CLINK) from control store;

3) Reading in the address of the first output gate (z);

4) Reading in the gate info word of the first output gate (z);

5) Reading in the address of the second output gate (z);

6) Reading in the gate info word of the second output gate (z).

Steps 3-6 depend on there being 2 outputs per gate. Referring back to Figure III-3, and using the gate whose info word is at m+9 as the x gate, step 1 reads the info word from address m+9 into a local store register. Step 2 reads the CLINK word at address m+11 into a local store register. This register contains the address n. Step 3 reads the contents of address n into another local store register. This register now contains the address (m+15) of the first output gate. Step 4 reads the gate info word for the gate at location m+15 into a register. Step 5 reads the contents of location n+1 (m+6) into a register. Finally step 6 reads the gate info word for the second output into a register. Thus the minimal x processing consists of steps 1-6. Now to estimate the timing on this, assuming best possible case, we will consider the time necessary to do the 6 reads. We assume address formation takes no time. Based on the timing constraints for control store [1:36], if we set up for the control store read in Tn, (assuming all T-steps are non-stretched and correspond to 1 T-period) the READ CS cannot legally occur until Tn+2. In the best case, we can also set up for a new read of control store in Tn+2, which produces the timing sequence below:

T_1 set up for read of x gate info word

T_2

T_3 read x gate info word, set up for read of CLINK

T_4

T_5 read CLINK, set up for read of first output address

T_6

T_7 read first output address, set up for read of first z gate info word

T_8

T_9 read first z gate info word, set up for read of second z address

T_{10}

T_{11} read second z address, set up for read of second z info word

T_{12}

T_{13} read second z info word.

Thus, the best case for x processing is 13 T-periods per x. Now let us examine the z processing. In the non-null case, we used one nanoword to do the setting of bits, etc., necessary in processing a z. We now need to add in the time necessary to link the gate into the MLINK, LINK queue. A measure of this task can be gleaned from the MULTI instruction ENQ. ENQ is an enqueue instruction designed for creating linked lists. It takes 27 T-periods [2:60]. We could possibly do better by using an ST (store) instruction of MLINK into the new gate's LINK and then an MVR (move register) of the new gate's address into MLINK. This approach requires 7T for the store and 5T for the MVR [2:54-55] for a total of 12T. We might further assume that custom nanocode could speed this up by 1/3 for a time expenditure of 8T. (8T is very close to the time necessary for this operation in the actual implementation.) Based on these new numbers, we can calculate a new lower bound on slow-down factor:

x processing time = # x gates processed X 13T/x gate

= 100 gates X 13T/gate

= 1300T;

null z processing time = # null z gates X 2T/gate

= 100 gates X 2T/gate

= 200T;

non-null z processing time = # non-null z gates X (5.4T/gate + 8T/gate)

= 100 gates X 13.4 T/gate

= 1340T;

total processing time = 1300T + 200T + 1340T

= 2840T.

This translates to:

time = 2840T x 80ns/T

= 227200ns = 227.2μs

slow-down factor = $\frac{227.2\mu s}{0.1\mu s}$

= 2272:1 slow down (for a .1μs machine cycle)

This figure is much more realistic than the 600:1 figure obtained before, but it is important to note that our inclusion of T-periods for processing in this analysis does not begin to approach what is necessary in the actual algorithm.

We propose to iterate through the calculations one final time, developing motivations for additions to the time estimates we have presented and ultimately defining a realistic best case estimate of the time required for performance of the algorithm.

To begin the analysis for this last iteration, we will modify somewhat the allocation of timing between the x and z processing. By this, we mean that the stepping down the output list and reading of the output gate info word is not really a function of x processing but belongs more properly in z processing. We will shift it into z processing for one primary reason. The x processing pipelined read of the outputs in the last analysis is not practical and really cannot be done in that fashion. The practical implementation is: read of one output; process the output; then loop back and read the next output. So, in the first step, we have taken 10 T-periods out of the x processing. (Time for read of CLINK and each of the output addresses and info words.) At this point x processing consists of reading only the x gate info word and requires 3 T-periods per gate.

As you will remember, the 3T estimate assumed that address formation took no time. In actual fact, if we assume that the address is in a local store register, address formation only takes the time necessary to set up the busses to use that as the control-store address. This adds 1 T-period. Thus to read the gate info word for an x requires 4 T-periods.

The next thing we need to do for x processing is to use the QM/1 micro-instruction execution capability to do a multi-way branch based on the data in the info word. Since we want to branch on more than the 7 bits available in the QM/1 local store register 31 C-field, we need some extra processing to set up the proper address. This processing, plus the multi-way branch itself, requires 5 additional T-periods. (The minimum nanocode segments are given in Appendix C.) Thus the basic x processing set up takes 9 T-periods. Based on our implementation, the actual x processing takes from 2 T-periods (for a gate not properly queued; i.e., no action necessary) to 9 T-periods for a gate requiring more complex processing. The time for the most standard processing (gate normally queued) is 5 T-periods. Thus for each x: set up, multi-way branch and x processing takes 9T + 5T = 14T. The only remaining step is to set up for processing of z's for each x and the set up (address formation) for processing the next x in the queue.

The set up for z processing consists of calculating the address of the CLINK word for this gate (gate info word address +2) then reading in CLINK to get the address of the output list. This processing takes 6T. The end of x processing for the current gate consists of setting up for the next x. This involves calculating the address of LINK and then reading the value of LINK. This takes 6 T-periods. Thus, the total x processing is given by:

$$\begin{aligned} x_{total} &= x_{set\ up} + x_{branch} + x_{proc} + x_{z\ set\ up} + x_{next} \\ &= 4T + 5T + 5T + 6T + 6T \\ x_{total} &= 26T \end{aligned} \qquad (6)$$

Z processing consists of setting up for the processing of the current output, doing the actual processing, and doing the preliminary set up for processing the next output. The first part consists of reading the address of the gate info word for this output and then reading the gate info word itself. This is then followed by the multi-way branch (similar to the x processing multi-way branch). This operation takes 12T (see Appendix C).

Actual z processing takes 2T for the null case, and from 5T to 18T for the non-null case. To this 5-18T we need to add the time necessary to add this z to the queue. This time is 6T. Thus, for non-null z processing, using the most common case of 7T for processing plus 6T for the queue addition, we need 13T. So, for z processing itself we have:

null z processing = 2T;

non-null z processing = 7T + 6T = 13T.

The final set up for next z is essentially included in the set up for this z. The only thing that is not done is the testing if this is the last output. We will assume the sign bit in the last output is set to 1. The time required to do this test is 4T if it is the last gate and 8T if it is not. (We will use 6T for our figures based on an average fan-out of 2.) Thus, the total z time is:

$$z_{total} = z_{set\ up} + z_{process} + z_{next}$$

$$= 12T + 2T + 6T \quad \text{(null z)}$$

$$= 20T \text{ (null z)} \tag{7}$$

$$= 12T + 13T + 6T \quad \text{(non-null z)}$$

$$= 31T \text{ (non-null z).} \tag{8}$$

We now have all the figures necessary to calculate best case slow-down. As an aside, please note that the nanocode given in Appendix C will not work if put together. The most striking example of the reason for this is the processing to determine if this z was the last in the output list. Remember we assumed the sign bit was set. This means that when we read the gate info word, we would have to clear all sign bits before the read. This is not accounted for in the nanocode of this example. There is also no provision for testing the last x in the queue. But as a best case timing estimate, these figures define the range of numbers involved. So, the calculation of slow-down factor looks like:

x processing time = # x gates processed X 26T
= 100 gates X 26T
= 2600T;

null z processing time = # null z gates X 20T
= 100 gates X 20T
= 2000T;

non-null z processing time = # non-null z gates X 31T
= 100 gates X 31T
= 3100.

This translates to:

time = (2600T + 2000T + 3100T) X 80ns/T
= 7700T X 80ns/T
= 616000ns = 616μs.

Thus for a .1μs cycle machine, the slow-down factor is 6160:1.

In summary, it is obvious that there are several parameters which determine the slow-down factor for a given case. The parameters are:

1. system size in total gates;
2. number of gates in system which change value during a cycle (average). This may be expressed as a percentage of system size;
3. average fan-out per gate;
4. cycle time of the emulated system.

For our analysis here we assumed that:

1. system size = 2000 gates;
2. percentage of gates changing = 100 gates = 5%;
3. average fan-out = 2;
4. cycle time of the emulated system = 0.1μs.

In the following sections, we present some details of the algorithm we implemented and the results of our timing studies. Since those timing studies address a 1μs cycle time machine (10 times slower than the machine we assumed here) we can recalculate the slow-down for our idealized implementation. It then becomes 616:1. Remember that this does not take into account all of the necessary actions. It is thus reasonable to expect that the slow down factor for a 1μs machine to be best case 600-800:1 and for a .1μs machine to be 6000-8000:1. In summary, for the slower machine, with a smaller system (2000 gates), the LRC estimate of 500-600:1 is quite optimistic but still a reasonable figure.

IV. ADDITIONAL FEATURES OF THE ALGORITHM

The algorithm introduced in the previous section includes additional elements which allow it to handle the types of situations expected in real-world applications. Aside from handling all types of gates in the same manner, and being able to quickly process gates without having to review every input of every gate, this approach takes into account the possibility for double queueing. If two gates share the same output gate and both change value such that the output gate should change, they will both independently queue the same gate for processing. If this happens within a single logic level, it is quite possible that the common output gate should in fact not be queued at all, since as a result of both inputs its value should remain the same. To handle this sort of case,the NASA LRC algorithm includes processing which prevents unnecessary queueing, as well as a second set of flags (V_2 and Δ_2) and a second queue linkage word ($LINK_2$) which are used as a means of remembering the necessary data for processing an additional queueing if in fact one is required. This latter situation arises when double queueing is spread over two propagation cycles (two logic levels).

An additional feature of this approach is that flip-flop devices are treated as ordinary gates with some additional special case considerations. The flag FF is used to indicate a flip-flop device and enables the algorithm to handle such a device effectively. Involved in this process is the flag T, which indicates a trigger input for a flip-flop, requiring a slight variation in treatment.

A list of the variables involved in this algorithm and their usage is provided in Table IV-1, and a section of the algorithm dealing with

double-queueing is detailed in Figure IV-1. It is entered only if during the normal processing of an output z-gate, that gate's CNT value transitioned into or out of zero. (Thus indicating that this z gate should be queued for future x gate processing.) The variables of most concern here are: 1) the "properly queues" flag Δ_1, which indicates that a gate is queued for x processing, and when needed for double queueing enables a gate to be "dequeued" without actually dequeueing the gate itself; 2) the "cycle queued" flag Δ_3, which remembers in which propagation cycle (C) this gate was queued for x processing (propagation cycles indicated by C are equivalent to x queue processing cycles, and each represents the processing of one logic level of the system); 3) the current value of the gate V_1; and 4) the linkage variables $LINK_1$ and MLINK used in the linked-list x queue whose ties between gates define the course of any given processing cycle.

The variables concerned with the queueing of a gate onto the second x queue include V_2, Δ_2, Δ_4, and $LINK_2$. Although we implemented the algorithm as given to us, we feel there are some functional discrepancies involved in the manner in which these variables are used. The concept presented here, however, is of more importance than the details of its design in the algorithm. The need being addressed here concerns the queueing of gates a second time. (This situation arises when double queueing occurs over two consecutive propagation cycles, as discussed briefly above.) The idea is to remember what the value of the gate is at the time of the second queueing, in order to correctly process the gate when its first processing cycle begins; and to queue the gate properly for a second processing. The

area of the flow in Figure IV-1 encircled with dashed lines attempts to accomplish these goals. The flag Δ_4 indicates that the second queue is employed for this gate; Δ_2 is later to become the Δ_1 of the second processing cycle, and indicates that proper queueing has occurred; and V_2 remembers the current newly changed value of the gate. (The second queue linkage involving $LINK_2$ and MLINK in the given algorithm does not work properly when integrated with the normal linkage system using $LINK_1$.)

These additional features do cause extra overhead in the execution of the algorithm, but they enable the algorithm to emulate a wider range of real-world systems and to accommodate all the currently foreseeable events which occur in gate level emulation.

Variable	Definition / Usage
Δ_1	"Properly Queued" Flag: Indicates Gate Queued For X Processing.
Δ_2	Δ_1 For Second Queue.
Δ_3	"Cycle Queued" Flag: Indicates Value Of C When Gate Queued For X Processing.
Δ_4	Flag Indicating Gate Queued Onto Second Queue.
V_1	Current Output Value Of This Gate.
V_2	Gate Value For Processing Of Second Queue.
FF	Flag Indicating A Flip-Flop Device.
T	Flag Indicating A Flip-Flop Trigger.
CLINK	Pointer Word Containing The Address Of The Output List For Each X Gate.
$LINK_1$	Linked-List Linkage Word Pointing To The Next Gate In The Queue (Zero If Last Word In The Queue).
$LINK_2$	$LINK_1$ For Second Queue.
MLINK	Pointer To First Gate Of The Next Queue (Each X-Queue Cycle Starts A New Queue), And Zero If End Of Machine Cycle.
C	Propagation Cycle (X-Queue Cycle) Indicator (Alternates Value For Each Logic Level Procesed).

TABLE IV-1: Definition Of Variables

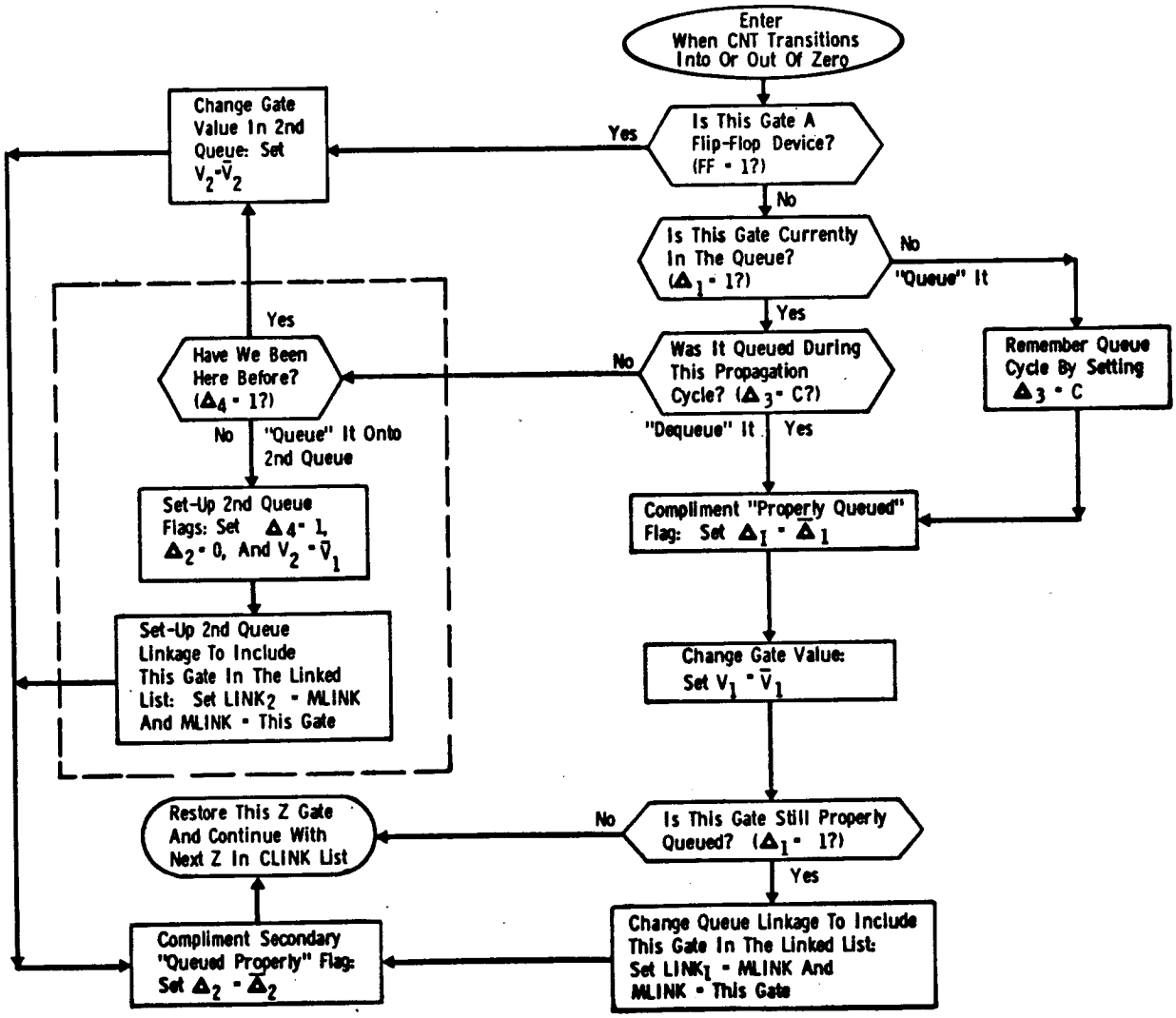


Figure IV-1 Flow Of Double Queueing Logic

V. IMPLEMENTATION OF THE ALGORITHM

In implementing the NASA LRC algorithm previously described, we have used the unique micro-instruction decoding capabilities of the QM-1 as a means of efficiently handling all of the individual flag conditions which arise in the course of normal processing. It is important to recognize in this algorithm the inherent dependence upon individual flag-bits and the large amount of processing necessary to handle them properly. Conventional coding methodology requires these flags to be tested and manipulated individually (which can be quite burdensome). A great deal of speed and flexibility can be gained by combining all of these flag-bits into one n-bit computer word, and subsequently using this word as the address of a specific routine in memory written to handle the exact bit pattern found in that arrangement of flag-bits. Thus we associate one word of data with each gate in the system, and we arrange that word so that each bit is dedicated for use as a specific flag. Then when the value of a flag-bit is needed to be known in order that some action may be taken, rather than reading each bit and testing for one or zero, the entire set of flag-bits is taken together as a "condition set" and used as the absolute address in nanostore of the routine which performs the exact actions necessary under the conditions specified by the flag-bits. In addition, when processing of that gate changes one of these flags, the appropriate bit of that gate's "info word" is changed to reflect the latest condition of the flag. This is very fast and very effective, but it does require a great deal of memory (in this case, nanostore). For our use, however, this drawback is far outweighed by the execution speed and flexibility gained.

Figure V-1 gives an overview of the logic used to implement the NASA LRC algorithm. The boxes containing an asterisk (*) or asterisks (**) include

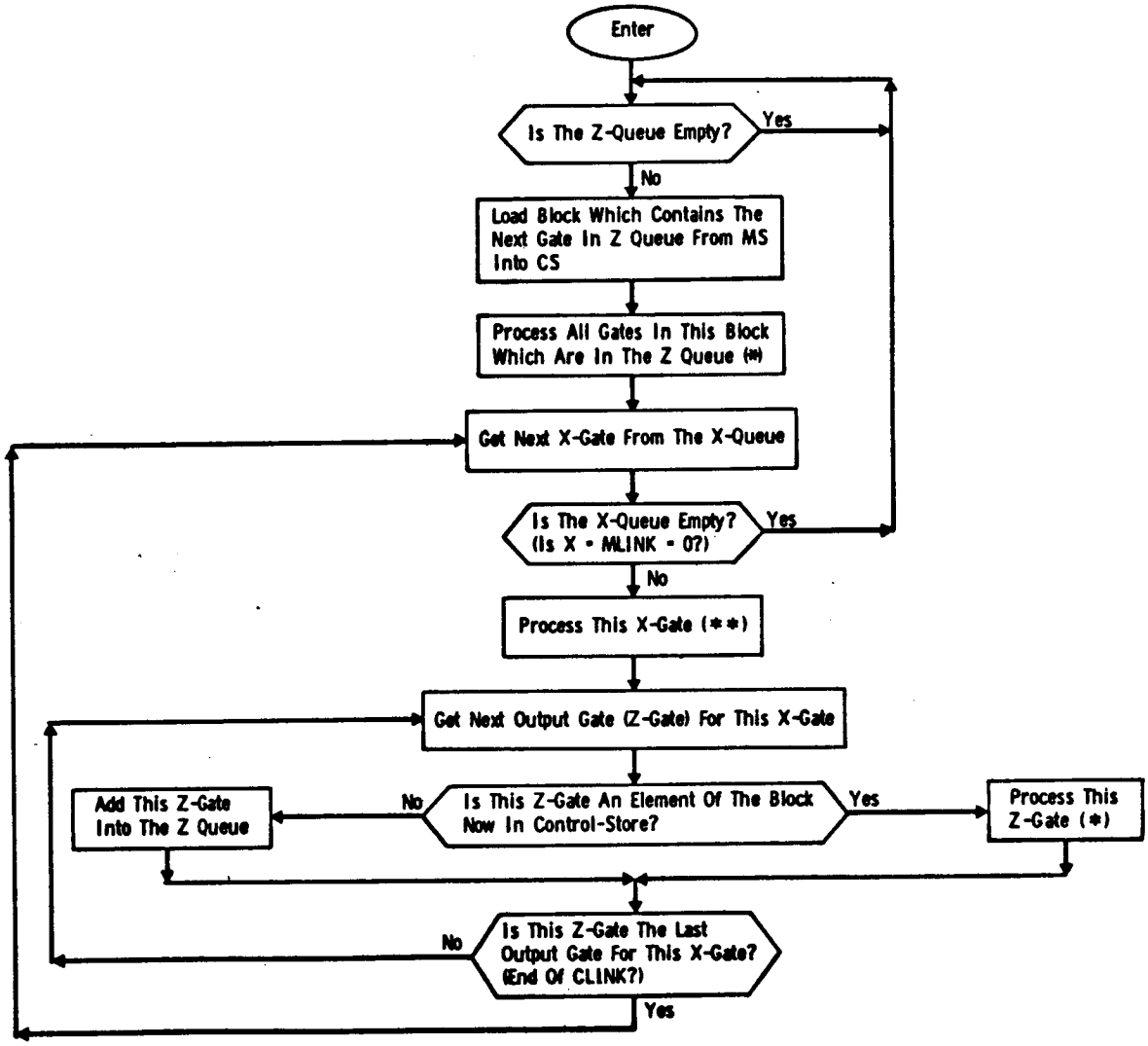


Figure V-1 Overview Of Implementation Logic

the type of processing described above. Before detailing these however, it is first appropriate to examine the lay-out of the memory tables used and the reasoning behind their structure.

Figure V-2 shows the structure of the "blocks" of gate data which reside in main-store and are loaded one at a time into control-store for processing. (Note that for an all control-store resident system no loading of blocks nor pre-x-cycle z-gate processing is needed.) In order to handle the most general case of system size and design, we designed our emulation to handle systems larger than 6000 gates (which is the largest system under this design to be totally control-store resident). This is accomplished through the use of the block structure and the means to keep track of interblock connections as follows. For each gate there are four 18-bit words which are reserved for dedicated use. The first word is the "gate info word" containing the flag-bits (detailed in Figure V-2) and the CNT counter for this gate. It is the lower three bits of this word, concatenated with the upper seven bits of the same word, which comprise the 10-bit nanostore address used for branching to the various processing routines as described earlier. It is of importance to point out here that the three bits FF, T, and the "z or x" flag, are used as a "nanostore page address" and therefore must be placed into F-register FIDX prior to branching to any routine. This enables the use of the micro-instruction decoding facility of the QM-1, which concatenates the 3-bit page address found in FIDX with the 7-bit address found in the C-field of local store register 31, to form a 10-bit absolute nanostore address. Thus the decoding of all flag-bits for each gate can be done "instantly" by

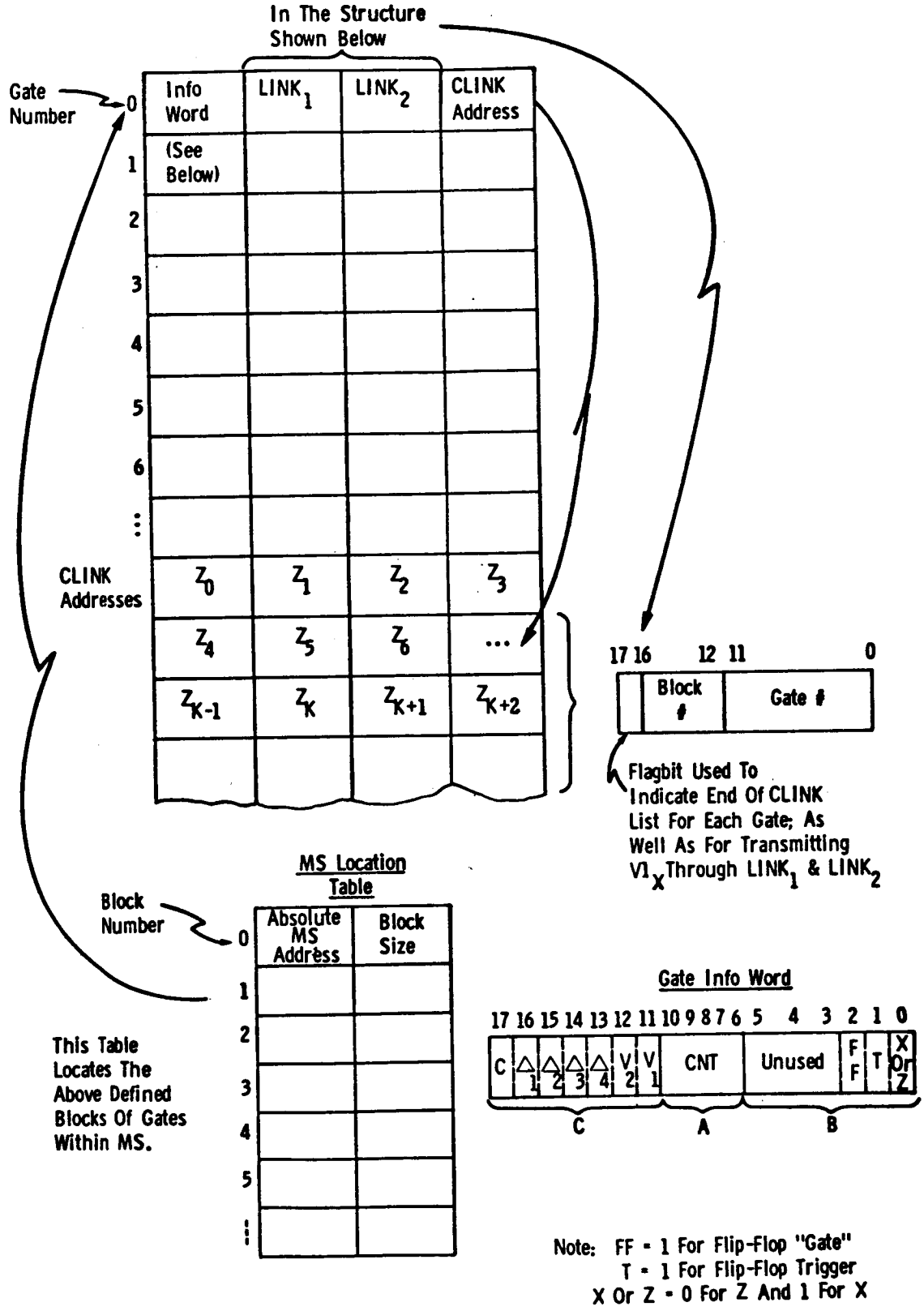


Figure V-2 BLOCK STRUCTURE IN MEMORY

placing the gate info word into R31, and the B-field of R31 into FIDX, and then invoking the micro-instruction decoding facility. This causes a branch to a dedicated routine which sets/resets the flag-bits as necessary for its exact input conditions, and then returns to a common continuation location in the main processing routine.

The second and third words of the gate block structure are labeled $LINK_1$ and $LINK_2$. These are used as linkage address words in the linked-list x-queue structure which controls the flow of x processing. ($LINK_2$ is used only for secondary queueing as discussed in section IV). They contain the block number and gate number of the next gate in the queue.

The fourth word is labeled CLINK and is the relative address of the start of the list of output gates (z-gates) associated with this x-gate. The number of output gates for each x-gate will vary from gate to gate of course, but for sizing considerations in this study we have averaged the fan-out factor at 2.0 output gates/x-gate. Thus we need 6 words per gate in each block (the four words described above and one word for each output gate). This gives an average size of 36000 words/block for a block size of 6000 gates.

Because the blocks will generally vary in size, there is an additional table in main-store dedicated to locating blocks in memory. It is indexed by block number and contains the absolute main-store address of the start of each block and each block's size (as the number of 18-bit words in the block). This table is also shown pictorially in Figure V-2, and is particularly useful in aiding the process of block loading (from main-store into control-store and back again).

There is one more table of interest in this design. This is a control-store resident "free core pool" table which is dedicated for use as a linked-list queue. When, during normal x-gate processing, an output gate (z-gate) comes up for processing which does not reside in the same block as the x-gate, then it becomes necessary to queue such z-gates for future processing (when the appropriate block is loaded into control-store). This free core pool queue is used to queue these z-gates for "pre-x-cycle processing", and is termed the "z queue". It is depicted in Figure V-3.

Now, returning to Figure V-1, the first action is to process (pre-x-cycle process) the z-gates waiting in the z queue. So if the queue is non-empty (and since the gates are ordered sequentially by block number/gate number), the next z-gate in the queue dictates what block should be loaded from main-store. When the loading is completed, those z-gates in the queue which reside in this block are processed as normal z-gates in the following manner: The CNT counter for each z-gate is updated according to the value of the x-gate for which the z-gate is an output. (If $V_{lx} = 1$, CNT_z is incremented, and if $V_{lx} = 0$, CNT_z is decremented. For z-gates whose x-gate is in some other block, V_{lx} is passed as the sign-bit of the queue element itself, as shown in Figure V-2, similar to the "end-of-CLINK" flag-bit K.) If this CNT transitions into or out of the value zero, then the z-gate becomes "non-null" and a branch is taken using the micro-instruction decoding facility to the appropriate z-processing routine. Upon return from this routine, the z-gate has been queued into the x-queue (using $LINK_1$, etc.) for future x-processing. Those z-gates

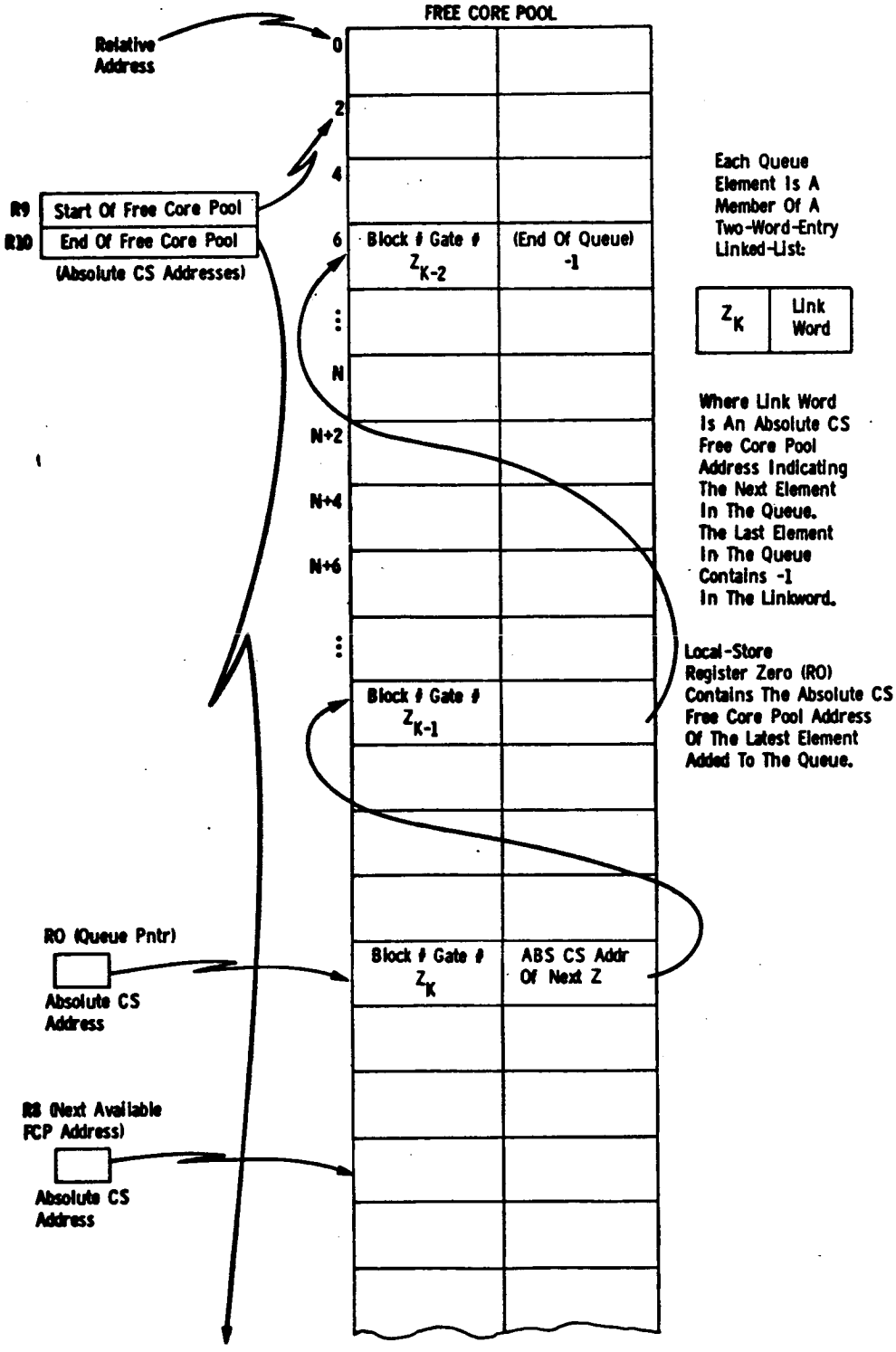


Figure V-3 Output Gate (Z) Queuing

whose CNT does not transition into or out of zero need no other processing, so they (and the returned non-null z-gates) are simply restored into the control-store block for future reference. This is the procedure for normal z-processing and occurs in Figure V-1 in the boxes marked with an (*)

Following the pre-x-cycle z-processing, we begin normal x-processing. This includes the two loops and the same basic logic shown in Figure III-1, with the actual x-gate processing (shown as the box marked with (**) in Figure V-1) accomplished via the micro-instruction decoding facility. Note an additional difference exists here, in that z-gates to be processed for a given x-gate must be checked first to see if they reside in the current block in control-store. If they do not, they are added into the z queue for future processing. And if they are in this block they are processed as described above for normal z-gate processing.

The implementation of this algorithm has been coded in nanocode, and the timing studies discussed in section VI and Appendix B for non-control-store resident systems are entirely based upon this implementation. For systems which are completely control-store resident (system size $\leq$ 6000 gates), the timing studies presented are based upon the normal x and z processing loops of this implementation, skipping the sections of code dealing with z queueing. This is reasonably accurate as an estimate of resident system timing considerations. However, the linkage words and CLINK "addresses" in memory are still in "block number/gate number" form, and hence incur additional unneeded overhead for absolute address computation, etc. If this code were optimized to be a strictly control-store resident emulation (instead of the generalized "handle all cases" emulation now coded), the efficiency of processing could be significantly increased, and perhaps a 20%-30% timing improvement realized.

VI. TIMING RESULTS

In order to gain an understanding as to the applicability of the algorithm described in the previous sections, we derived two equations which enable the generalized projection of the timing factors involved for various kinds of systems. (The derivations of these relations and examples of their use are presented in Appendix B.) As an initial, "most simple" case, we examined systems which reside totally in control-store (and therefore need no data storage external to control-store nor the associated loading and linkage software.) The amount of time necessary to emulate a single machine cycle is given in T-periods by the relation:

$$T_{resident} = (68 + 35F)x + 29y$$

where F is the fan-out factor defined to be the number of output gates per gate processed (or the number of z gates per x gate); x is the number of gates changing value in the system; and y is the number of queue processing cycles needed to emulate the data propagation associated with a complete machine cycle. This can be thought of as the number of logic levels in the system.

Figure VI-1 displays this equation plotted for F = 2.0 outputs/gate, for the value of x ranging through 5%, 10%, 15%, and 20% of the system size. The datum of most interest here is that a single cycle for 6000 gates is emulated at a slow-down factor of 3,492:1 for a real machine cycle of 1μs, with 5% of the system changing. Notice the effect of changing F from 2.0 to 3.0 outputs/gate in Figure VI-2. A single cycle for 6000 gates at 5% changing now results in a slow-down factor of 4,322:1 for a 1μs machine cycle. This change from a fan-out factor

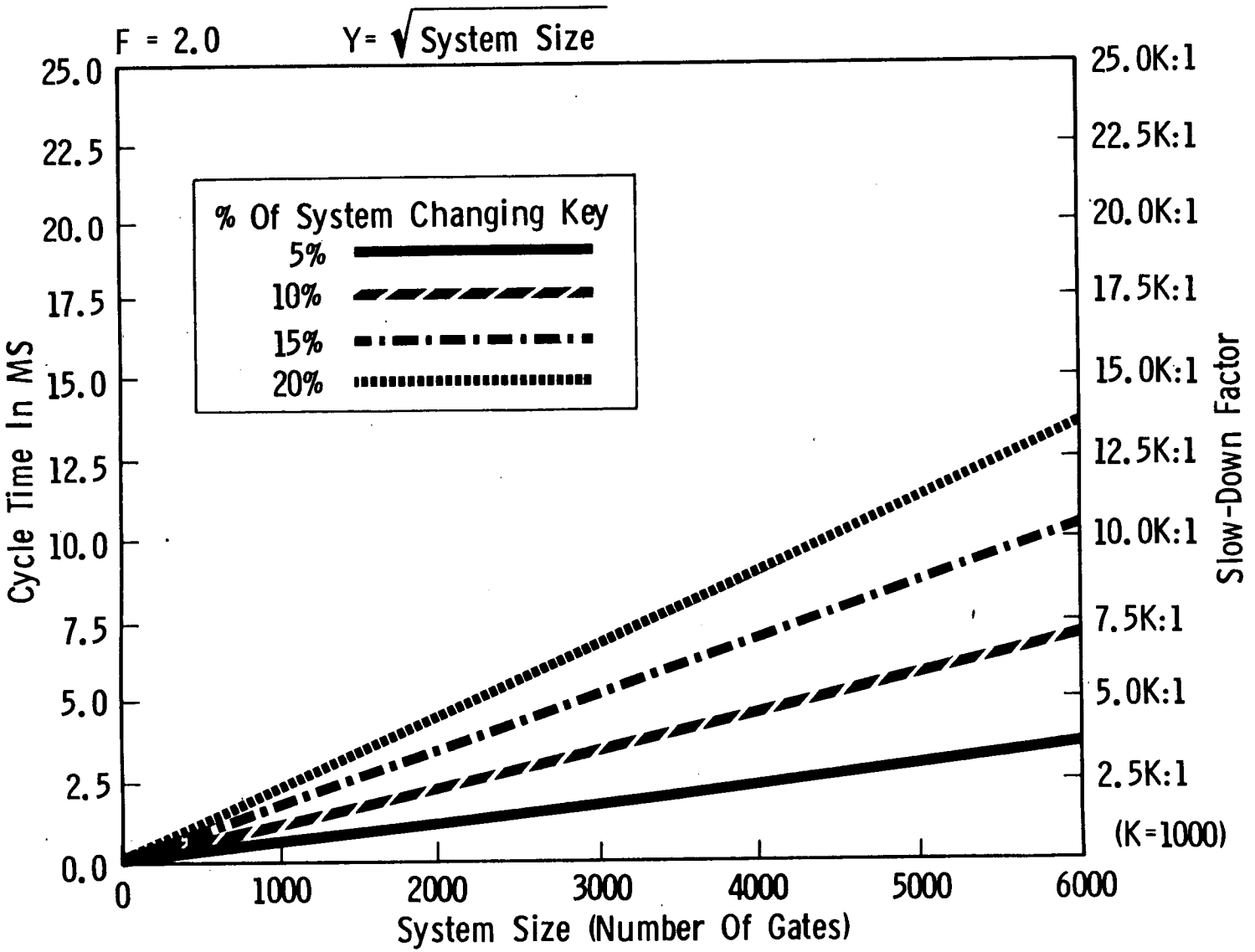


Figure VI-1 Emulation Cycle Time For Control-Store
Resident System: T=(68 + 35F) X + 29 Y

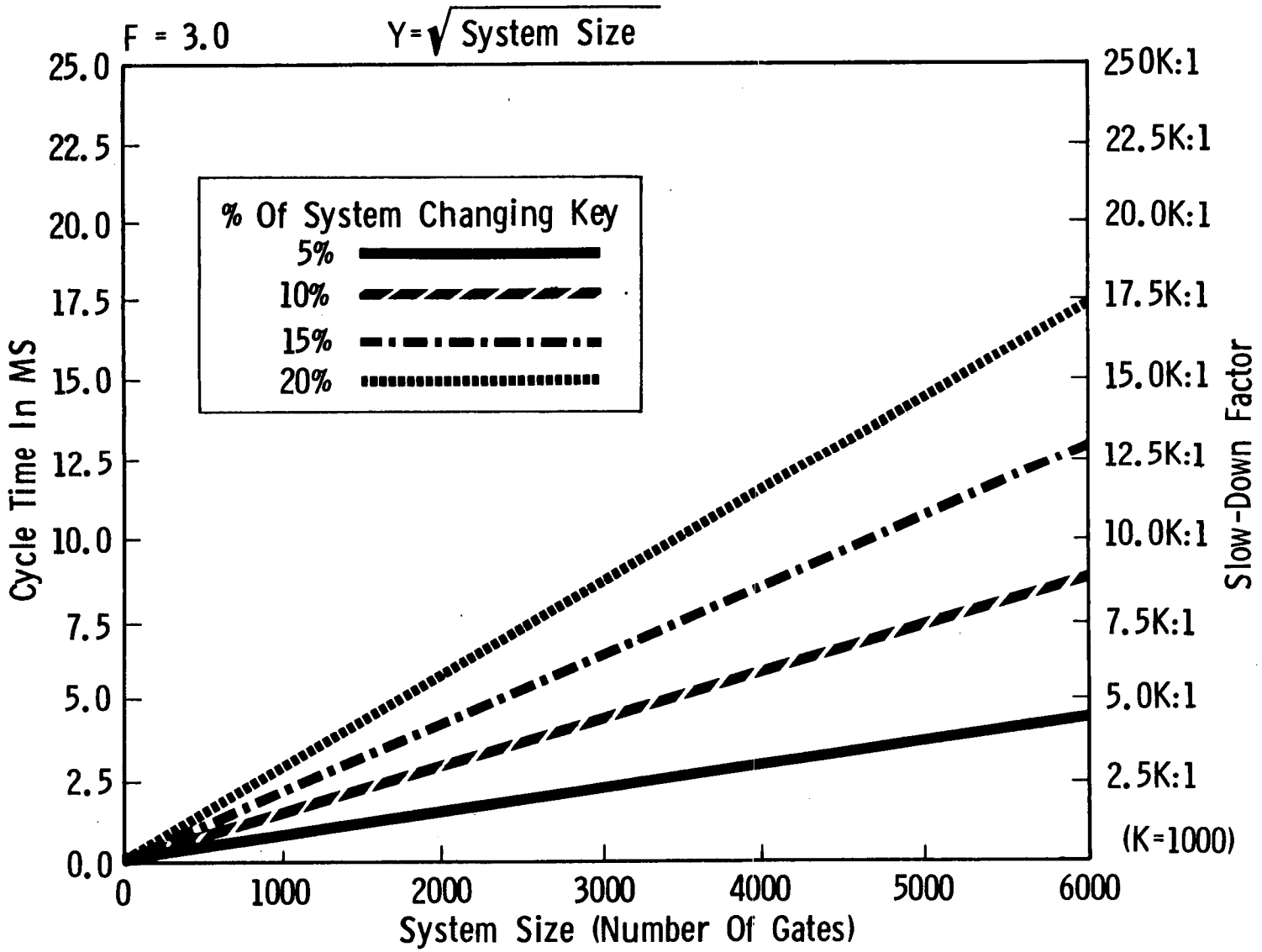


Figure VI-2 Emulation Cycle Time For Control-Store Resident System: T=(68 + 35F) X + 29 Y

of 2.0 to 3.0 outputs/gate generally results in a 20% increase in slow down factor. The data points for these curves are given in Table VI-1.

We can gain a more comprehensive understanding of how these curves relate to one another by translating them into "sample time" (i.e., the amount of time needed to emulate .1 second of real time execution on a 1μs machine). Figure VI-3 shows the curves for F = 2.0 and 3.0 both for 5% of the system changing and for 20% of the system changing. Notice that a single sample for a 6000 gate system at 5% changing and F = 2.0, requires 5.82 minutes, where the same sample at 20% changing requires 22.38 minutes. (Further data points are given in Table VI-2.) We can also see that changing F from 2.0 to 3.0 increases the sample time by the same 20% seen above.

It is perhaps most useful to view this data from an experimental point of view, and to see how long it would take to run for example 10,000 samples of .1 second real time each. Figure VI-4 displays this information. For a 6000 gate system with a machine cycle time of 1μs, with 5% of the system changing, it takes approximately 40 days or about 1 1/3 months to run a 10,000 sample test with F = 2.0. The data points for these curves are given in Table VI-3.

In addition to the above described "most simple" case, we expanded our study to include larger systems whose size necessitates system residence in main-store with "blocks" of gates being loaded into control-store for processing. The equation for these systems in generalized form is as follows:

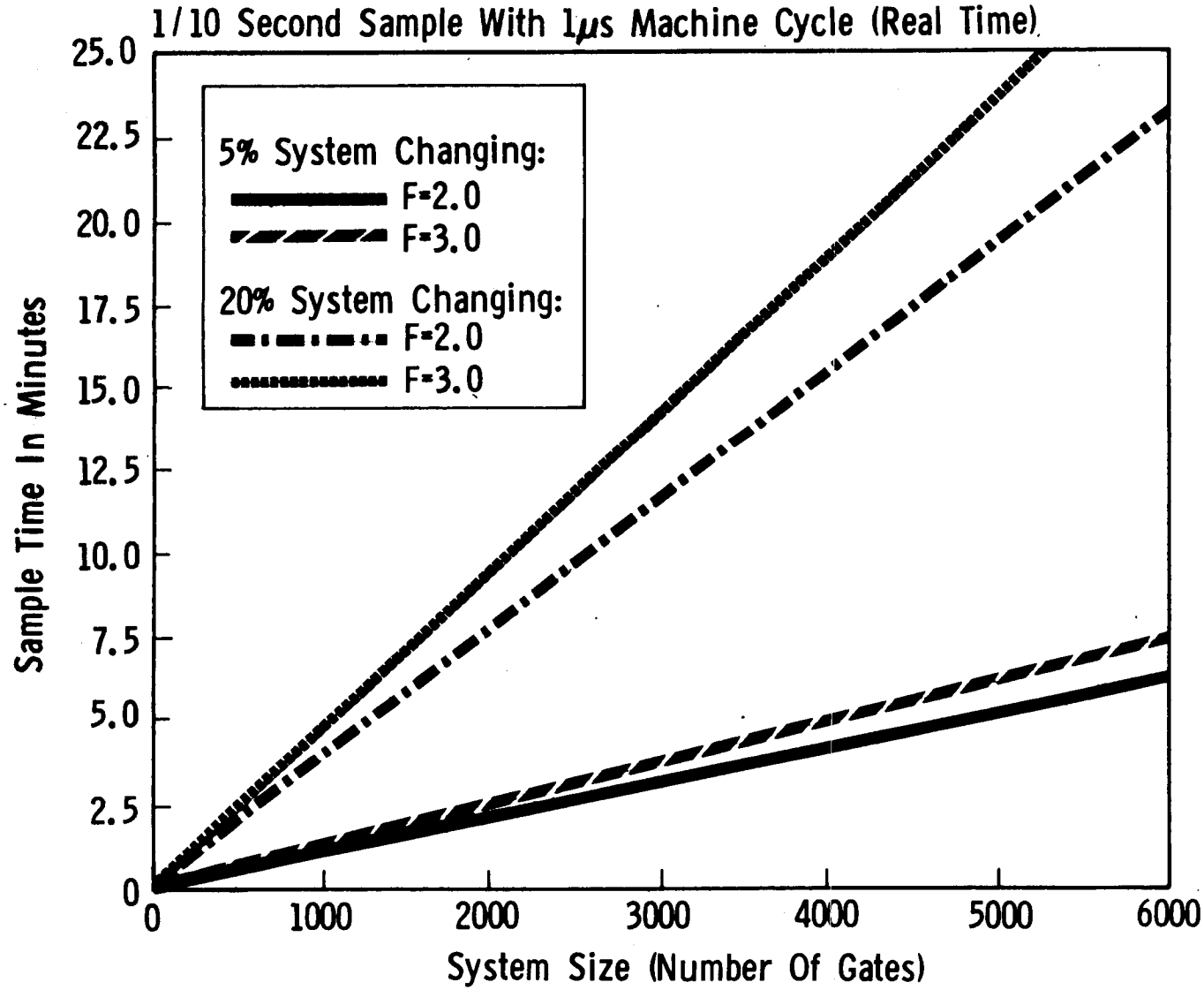


Figure VI-3 Emulation Sample Time For Control-Store Resident System

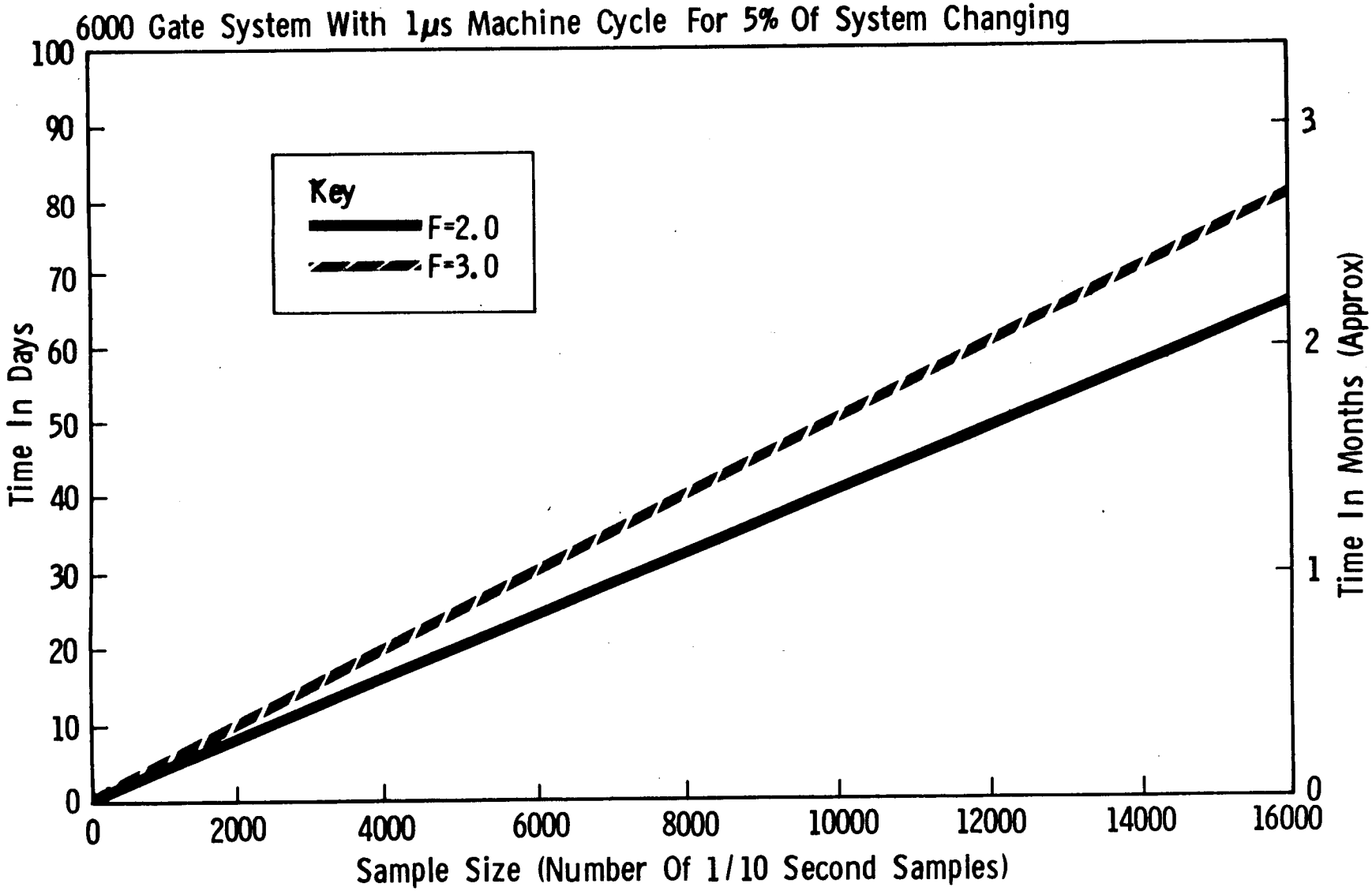


Figure VI-4 Emulation Time Per Sample Size
For Control-Store Resident System

$$T_{\text{non-resident}} = (\#\text{ blocks in system}) \times \{(44) \times (4 + F) \times (\text{block size}) + 84 \times (z \text{ prequeued}) + 94 + (z \text{ queued}) \times [72 + (25) \times (n - 1)] + (68 + 35F)x + 29y\}$$

where z prequeued, z queued, and n are variables associated with the processing of output gates which reside in blocks other than that of their input gates (see Appendix B for exact details). In order to gain an appreciation for the meaning of this relation and the processing involved in such an emulation system, we will consider a system with only two blocks, each block sized at the maximum available control-store, 6000 gates. Thus we have a 12000 gate system. Assuming 5% changing and F = 2.0, as above, we find that a single cycle on a 1μs machine takes 260,579,630ns. This is a slow down factor of 260,580:1. A single .1 second real time sample would take 7.238 hours, so a 10000 sample test would require 8.26 years! As you can see this is not a feasible approach. The reason these numbers are so high is that main-store accessing is extremely slow. It requires 22 T-periods/word to transfer from main-store into control-store, which means to load a single 6000 gate block (with 6 words of data required for each gate) it takes (22) X (6000) X (6) = 792,000 T-periods = 63.36ms. Hence this overhead becomes quite prohibitive.

Table VI-1a: Emulation Cycle Time (in MS)
For Control-Store Resident System

System Size / X=	1000 Gates	2000 Gates	3000 Gates	4000 Gates	5000 Gates	6000 Gates
5% Of System Changing	.6254	1.2078	1.7830	2.3547	2.9241	3.4917
10% Of System Changing	1.1774	2.3118	3.4390	4.5627	5.6841	6.8037
15% Of System Changing	1.7294	3.4158	5.0950	6.7707	8.4441	10.1157
20% Of System Changing	2.2814	4.5198	6.7510	8.9787	11.2041	13.4277
Y =	31.62	44.72	54.77	63.25	70.71	77.46

T In MS = (68 + 35F)x +29y For F = 2.0 Output Gates Per x-Gate

$Y = \sqrt{\text{System Size}}$

Table VI-1b Emulation Cycle Time (in MS)
For Control-Store Resident System

System Size / X =	1000 Gates	2000 Gates	3000 Gates	4000 Gates	5000 Gates	6000 Gates
5% Changing	.7654	1.4878	2.2030	2.9147	3.6241	4.3317
10% Changing	1.4574	2.8718	4.2790	5.6827	7.6841	8.4837
15% Changing	2.1491	4.2558	6.3550	8.4507	10.5441	12.6357
20% Changing	2.8414	5.6398	8.4310	11.2187	14.0041	16.7877
Y =	31.62	44.72	54.77	63.25	70.71	77.46

T In MS = (68 + 35F)x +29y For F = 3.0 Output Gates Per x-Gate

$Y = \sqrt{\text{System Size}}$

Table VI-2a Emulation Sample Time For A Single
1/10 Second Real-Time Sample On 1μs Machine

System Size / X=	1000 Gates	2000 Gates	3000 Gates	4000 Gates	5000 Gates	6000 Gates
5% Of System Changing	62.54 Sec 1.04 Min	120.78 2.01	178.30 2.97	235.47 3.92	292.41 4.87	349.17 5.82
10% Of System Changing	117.74 Sec 1.96 Min	231.18 3.85	343.9 5.73	456.27 7.60	568.41 9.47	680.37 11.34
15% Of System Changing	172.94 Sec 2.88 Min	341.58 5.69	509.50 8.49	677.07 11.28	844.41 14.07	1011.57 16.86
20% System Changing	228.14 Sec 3.80 Min	451.98 7.50	675.10 11.25	897.87 14.96	1120.41 18.67	1342.77 22.38

Control-Store Resident System With F=2.0

Table VI-2b Emulation Sample Time For A Single
1/10 Second Real-Time Sample On 1 μs Machine

System Size / X =	1000 Gates	2000 Gates	3000 Gates	4000 Gates	5000 Gates	6000 Gates
5% Of System Changing	76.54 Sec 1.28 Min	148.78 2.48	220.30 3.67	291.47 4.85	362.41 6.04	433.17 7.22
10% Of System Changing	145.74 Sec 2.43 Min	287.18 4.79	427.90 7.13	568.27 9.47	708.41 11.81	848.37 14.14
15% Of System Changing	214.94 Sec 3.58 Min	425.58 7.09	635.50 10.59	845.07 14.08	1054.41 17.57	1263.57 21.06
20% Of System Changing	284.14 Sec 4.74 Min	563.98 9.39	843.10 14.05	1121.87 18.69	1400.41 23.34	1678.77 27.98

Control-Store Resident System With F=3.0

Time vs Sample Size For
6000 Gate System With 5% Changing

F	1000 Samples	2000 Samples	4000 Samples	6000 Samples	8000 Samples	10000 Samples	12000 Samples	14000 Samples	16000 Samples	Time Per Sample
2.0	5820 Min 97.0 Hrs 4.04 Days	11640.0 194.0 8.08	23280.0 388.0 16.16	34970.0 582.0 24.25	46560.0 776.0 32.33	58200.0 970.0 40.42	69840.0 1164.0 48.50	81480.0 1358.0 56.58	93120.0 1552.0 64.67	5.82 Min
3.0	7220 Min 120.33 Hrs 5.01 Days	14440.0 240.67 10.03	28880.0 481.33 20.06	43320.0 722.0 30.08	57760.0 962.67 40.11	72200.0 1203.33 50.14	86640.0 1444.0 60.17	101080.0 1684.67 70.19	115520.0 1925.33 80.22	7.22 Min

(Each Sample = .1 Second Real Time With 1μS Machine Cycle)

Table VI - 3

VII. CONCLUSIONS

As we stated in the introduction, the intent of our implementation was to prove the feasibility of using gate level emulation technology in support of data collection for reliability studies of fault tolerant digital avionics equipment. It is clear that to support statistical measures, the key potential problem is the time necessary to execute a sample run on the gate level emulation. If this time is too long, the task of running a statistically significant number of samples becomes overwhelming. We have therefore focused our feasibility determination on the execution speed of gate level emulation.

As shown in Appendix B, our QM/1 implementation results in a 1200:1 slow-down factor for a 2000 gate, control-store resident system within the constraints given in that appendix. This datum is also shown in section VI in the graphs although it is not explicitly noted since the 6000 gate example given there is the limiting case. This 1200:1 slow-down compares very favorably with the best possible case for slow-down shown in section III of 600:1, since the implementation contains features which cause additional overhead for address calculation and system partitioning into blocks. The maximum resident system of 6000 gates also falls in the reasonable range of 3500:1 slow down.

On the other hand, the partitioned system case, of which the 12000 gate, 260,500:1 slow-down is an example, is clearly not feasible for any reasonable number of samples. Based on this, we conclude that the gate level emulations should be restricted to control-store resident subsystems, which for the QM/1 works out to a maximum of about 6000 gates. We do not feel this is overly restrictive considering that we have seen gate level simulations of current technology micro-processors which fall in the

range of 2000 gates. Thus 6000 gates can represent a fairly substantial subsystem. Furthermore, by restricting ourselves to completely resident emulations, further economies in the implemented algorithm can be achieved as mentioned in Section V. We estimate that we can achieve about a 20-30% improvement in speed.

The primary conclusion we can make based on the implementation for the QM/1 is that gate level emulation is feasible to do and provides the speed necessary for statistical studies of reliability.

Although the implementation we did was based on the QM/1 architecture, the restrictions imposed by that machine do impose a limit on what is achievable. Examples are the 6000 gate limitation and the additional overhead necessary to decode ten bits rather than the seven that the QM/1 is set up for. Three possibilities come to mind in terms of providing the emulation support capability for the final facility. The first of these is to consider making hardware modifications to the QM/1. This could include expansion of the maximum permissable control-store size or the addition of a bus to connect main-store and control-store directly. Secondly, other micro-programmable machines may be more amenable to the application. And finally, the possibility of building a special purpose, gate level emulation machine should be considered. Such a machine might be readily assembled from 2900 series chips. All three of these possibilities will be considered in the second phase of the contract.

VIII. REFERENCES

1. QM-1 Hardware Level User's Manual, Nanodata Corporation, March 1976.

2. MULTI Micromachine Description, Revision 1, Nanodata Corporation, March 11, 1976.

3. Digital Avionics Design and Reliability Analyzer, Feasibility Study Report, MCR-79-663, Martin-Marietta Aerospace Corporation, November 1979.

APPENDIX A

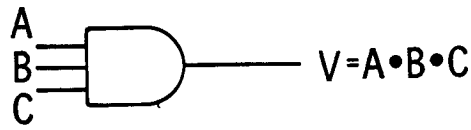
UNIFORMITY OF GATE TREATMENT

One of the primary benefits of the NASA LRC gate level emulation algorithm is the concept of gate processing independent of the function of the gate itself. What this means in practical terms is that the algorithm does not need to keep track of the gate type and can handle ANDs, ORs, NANDs, and inverters all in exactly the same fashion. This appendix is intended to provide a brief description of how this is possible by discussing a few examples to illustrate the processing done and decisions made.

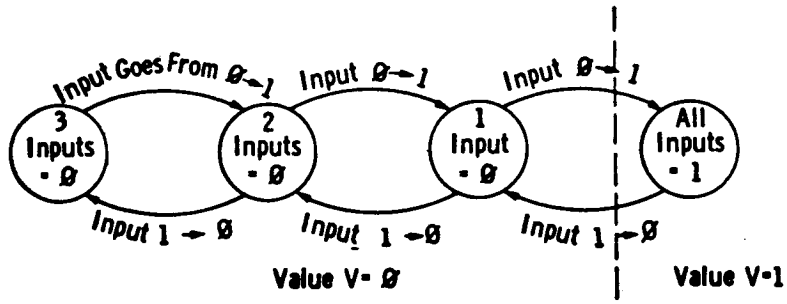
In order to process the gates, two values are required. The first represents the current value of the gate. We will call this V. The second quantity relates to the number of inputs of the gate. We will call this value CNT (for count). This number is the key to the processing and the distinction as to type of gate is characterized by the initial values assigned to CNT and V.

In operation, whenever an input to a gate changes value (from 0 to 1 or vice versa), the quantity CNT is operated on. For the 0 to 1 change, CNT is incremented by 1. For the 1 to 0 change, CNT is decremented by 1. The gate whose CNT is being updated will change value whenever CNT transitions either into or out of 0. That is, if either the old value of CNT is zero (before increment or decrement) or the new value of CNT is zero (after the increment or decrement), then the value V is changed (0 to 1 or 1 to 0 depending on current value). A few examples will best illustrate this.

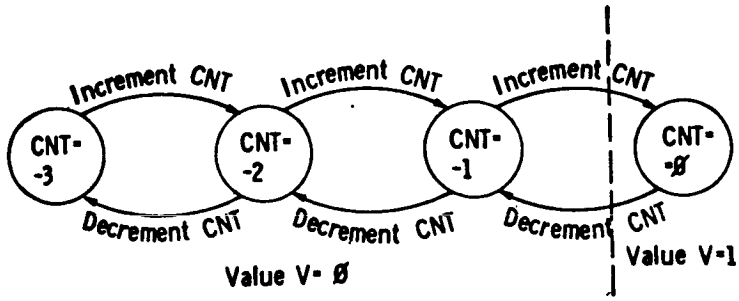
Example 1 : 3 input AND gate



The description of the action of the 3 input AND gate is best described by the following state diagram.



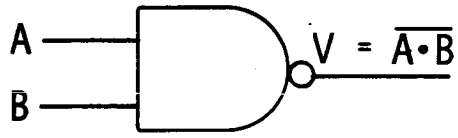
Note that the left to right arcs represent an input going from 0 to 1 while the right to left arcs represent an input going from 1 to 0. When we get to the rightmost state, all inputs are 1 and hence the output V is 1. In all the other states, at least one input is 0 and the output V is 0. Now suppose we let CNT = 0 for the case where all inputs = 1. The state diagram with CNT values in place of number of inputs = 0 is:



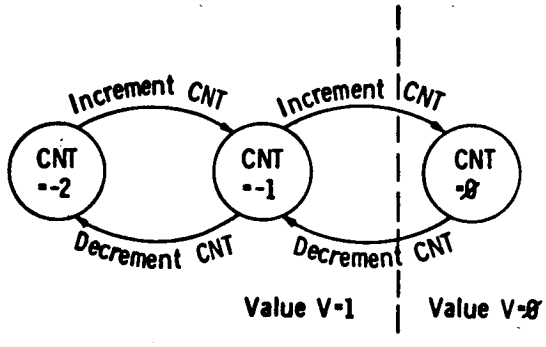
Note that the arcs on this diagram represent exactly the same as on the previous diagram; i.e., left to right is an input going from 0 to 1 and right to left is an input going from 1 to 0. The transition of

CNT into and out of 0 occurs across the dotted line and value V does indeed change when we cross this line. From this diagram, we see that, to initialize CNT and V for an n-input AND gate, we first assume all inputs are 0. We then set CNT = -number of inputs and V = 0. After initialization, we can blindly follow the specified processing and the proper gate output value will be produced.

Example 2 : 2 input NAND gate

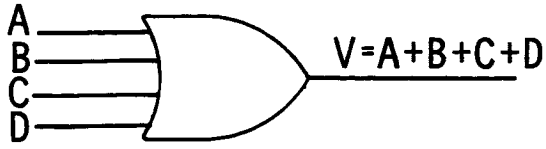


The NAND gate is a simple extension to the AND. The only difference is the value of V. V will be 1 to the left of the dotted line and 0 to the right. Thus a 2 input NAND gate state diagram looks like:

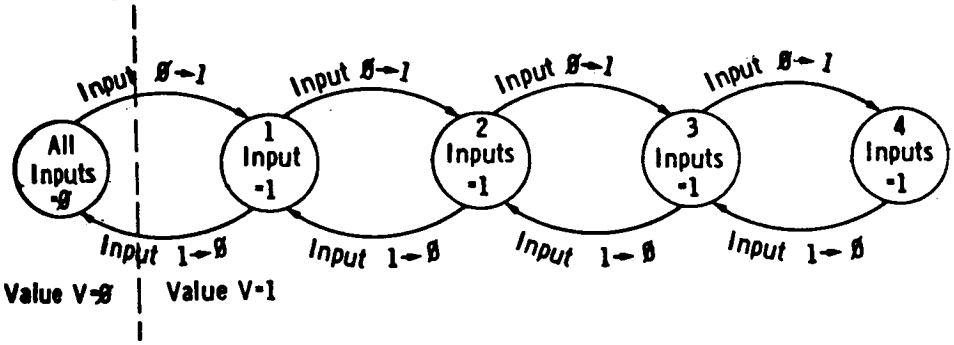


Initialization conditions are CNT = -number of inputs and V = 1.

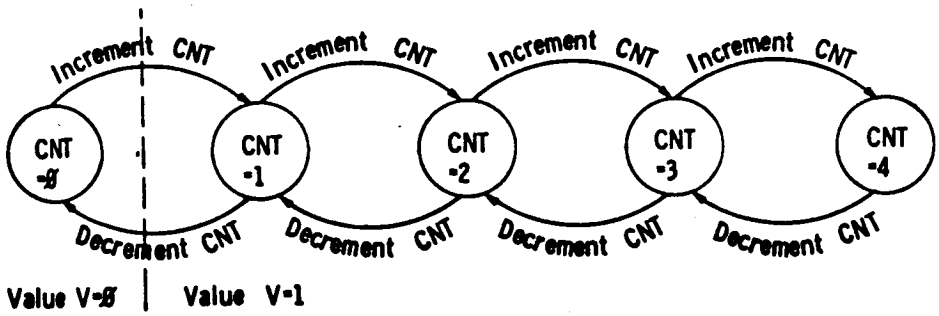
Example 3 : 4 input OR gate



The state diagram for the OR gate looks like:

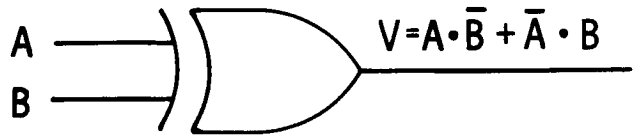


To isolate the CNT = 0 node in this case, we need to isolate the state to the left of the dotted line. Thus the CNT state diagram looks like:

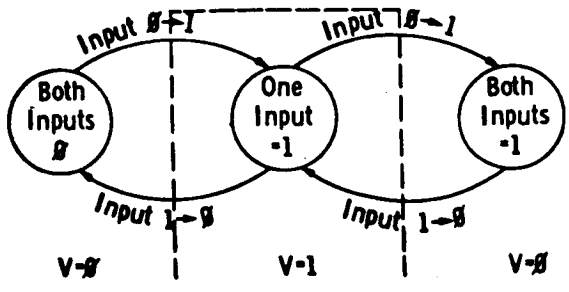


Thus the initial condition (all inputs 0) for an OR gate are: CNT = 0, V = 0. The extension to a NOR is made simply by using CNT = 0, V = 1 for initial conditions.

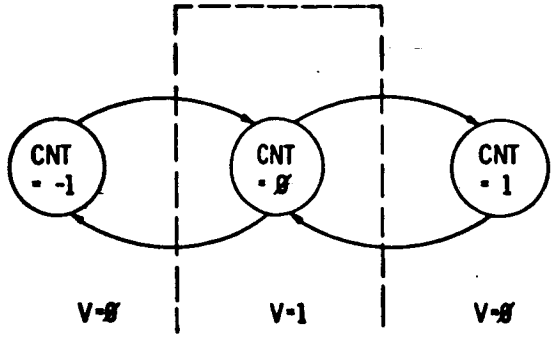
Example 4 : 2 input XOR gate



The state diagram for the XOR gate is:



The CNT state diagram becomes:



This again is a simple case for the general algorithm. Initial values for the two input cases are CNT = -1, V = 0. This is also the case for the general "odd" number of inputs type gate (i.e., 1 is produced for an odd number of inputs = 1). The only difference is that instead of signed arithmetic, modulo arithmetic is used (-1 mod 2 = 1).

Based on these examples, the initial conditions (assuming all inputs 0) for the most common gates are given in Table A-1. The inverter can be handled as either a one input NAND or a one input NOR.

Type Gate	Initial Value	
	CNT	V
AND	-number inputs	0
NAND	-number inputs	1
OR	0	0
NOR	0	1
INVERT	-1	1
INVERT	0	1
XOR	-1	0
NXOR	-1	1

Table A-1

Once the initialization has been done, the processing of each gate is exactly the same, regardless of type. In addition, the concept is flexible enough to be able to handle more non-standard type gates (e.g., the odd number of input counter which could be used for parity generation).

APPENDIX B

DERIVATION OF EQUATIONS

This appendix explains the basis for the system "emulation time" equations used in this report. Two basic equations are derived herein, the first for systems residing totally in control-store (e.g., system size $\leq$ 6000 gates); and the second an approximation for larger systems which necessarily have only part of the system in control-store and the remainder in main-store. This appendix first explains the former of these and then gives some examples. Following this is the derivation of the second equation and then an example of its usage.

It should be understood that these equations are derived from the actual implementation of the algorithm described in section V. The given timing considerations are simply the sum of the individual T-periods involved in executing the algorithm. (Thus when it is stated that x-processing takes 46 T-periods, this comes from examining the code itself. Recall that one T-period is the basic unit of time for nanocode, and is defined to be 80ns).

I. DERIVATION OF THE EQUATION FOR CONTROL-STORE RESIDENT SYSTEMS

Given: 1) x-processing requires 46 T-periods/x and 29 T-periods/logic level in the System (using 5 T-periods for most common case processing routine);

2) z-processing requires:

62 T-periods/non-null-z

and 35 T-periods/null-z

Where non-null-z's are those output gates whose counter (CNT) transitions into or out of zero as it is processed during normal z-processing. This results in that z gate being queued as an x-gate for processing in a future cycle. Null-z's are z-gates whose counter does not transition into or out of zero and hence cause no further action to be taken. In addition the last z processed for each x takes 5 T-periods less than the other z's, hence: -5 T-periods/x.

Adding these together we get

$$T_{resident} = (46 - 5)x + 62 \times (\#\text{ non-null z's}) + 35 \times (\#\text{ null-z's}) + 29y$$

with y = # logic levels in the system. (1)

Now consider that

1) the fan-out factor $F = \frac{\text{total \# z's}}{\text{total \# x's}} = \#\ z's/x$

and 2) that the # non-null z's is identically equal to the number of x's since each x comes from a non-null z.

Hence: $F = z/x$

(# null z's) + x = total # z's

so (# null z's) + x = xF

$\therefore$ # null z's = xF - x = x(F-1). (2)

Substituting equation (2) into equation (1) we get:

$$T = (46 - 5)x + 62x + 35[x(F - 1)] + 29y$$

$$= (41x + 62x + 35xF - 35x + 29y$$

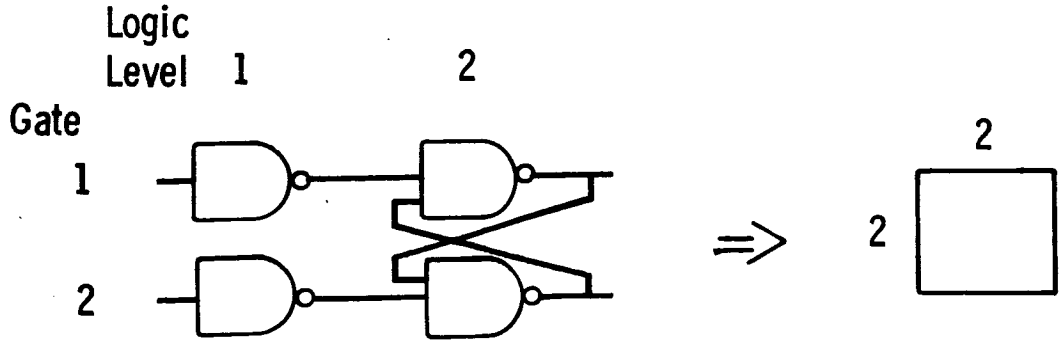
$$= 68x + 35xF + 29y$$

$$\therefore \quad T_{resident} = (68 + 35F)x + 29y. \qquad (3)$$

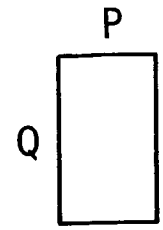
Equation (3) is the general form of the control-store resident system equation. Now for the following examples let us assume that 5% of the system changes at any given time. Thus

$$x = (.05)\ X\ (\text{system size}) \qquad (3.1)$$

Additionally, in order to estimate the value y, the number of logic levels (or x-queue cycles), we need to make an assumption concerning the system itself. To facilitate this assumption, we will deal with an intuitive concept called system shape. We assume the system is in general rectangular when the logic levels are plotted across the top of the diagram and the gates per level down the side. For example, the RS flip-flop below is square (2 logic levels w/2 gates/level)



Now, we recognize that, for larger systems, the general shape will be a non-square rectangle with the long side in the vertical direction; i.e.,



Where $P \leq Q$

However, the algorithm will translate feedback (as in the flip-flop shown before) as additional logic levels. Because of this, the general shape of the system, as seen by the algorithm, will become more square. Therefore we will assume for the following examples that the system is square so that the number of logic levels (y) = number of gates per logic level = $\sqrt{\text{system size}}$. **(3.2)**

Combining (3), (3.1), and (3.2) with a fan-out factor of

F = # z's/x = 2.0 we get:

$$T_{resident} = (68 + 35(2.0))(.05)(\text{System Size}) + 29\sqrt{\text{system size}}$$

Hence:

$$T_{resident} = (6.9)(\text{System Size}) + 29\sqrt{\text{system size}} \qquad \textbf{(3.3)}$$

<u>Example I</u>: System Size = 2000 gates:

$$T_{resident} = (6.9)(2000) + 29(\sqrt{2000})$$

$$= 13800 + 29(44.725)$$

$$= 15097 \text{ T-periods} = 1207760\text{ns}$$

Or a slow down factor = 1207.8:1

for a 1μs machine cycle. (12,077.6:1 for .1μs machine)

Example II: System Size = 6000 gates:

$$T_{resident} = (6.9)(6000) + 29(\sqrt{6000}\,)$$

$$= 41400 + 29(77.46)$$

$$= 43646 \text{ T-periods} = 3491680\text{ns}$$

Or a slow down factor = 3491.7:1

for a 1μs machine cycle. (34,916.8:1 for .1μs machine)

II. DERIVATION OF THE EQUATION FOR NON-CONTROL-STORE-RESIDENT SYSTEMS

In order to accommodate large systems whose size prohibits residence of the entire system in control-store, in this emulation "blocks" of gates (large tables of gate data) are loaded into control-store one at a time for processing, while the remainder of the system being emulated resides in main-store. When a block is loaded, the first processing necessary is that needed for z gates residing in this block which were queued by previous x gates in other blocks. This uses an additional queue, dedicated to this situation, and so we have termed this initial z processing "prequeue" processing. So we define "z prequeued" as the number of z gates processed in this prequeue phase. Similarly, "z queued" is the number of z gates queued during the normal processing of each block onto this dedicated z gate queue.

Now, given that:

1) it takes 22 T-periods/word to transfer 18-bit words from main-store into control-store (and visa versa), with an overhead of 28T-periods per block;

2) pre-queue processing of z's takes:

$$84 \times (z \text{ prequeued}) + 66;$$

3) there are 4 words/gate in the memory tables plus one word for each output gate (z). Assume a fan-out factor $\equiv \#\ z's/x = F$. Then $4 + F$ words are needed in memory per gate.

So to begin with, equation (4) below accounts for the loading of the new block from main-store into control-store, and the processing of z's which were queued by some previous block. All of this occurs prior to the normal x and z processing:

$$T_{pre} = (44) \text{ X } (4 + F) \text{ X (block size)} + 84 \text{ X (z prequeued)} + 94. \quad (4)$$

Equation (4) will be in effect for each block as it is loaded, and includes the restoration of each block to main-store.

In addition we must consider the processing of gates in the block while it is in control-store using equation (3).

Furthermore, subroutine ZQUEADD is used to queue z gates who reside in blocks other than the current block in core (onto the dedicated z queue). In this process each element in the queue is compared with the z being placed onto the queue to ensure sequential ordering of the queue (by block number and gate number). This searching takes 25 T-periods for each queue element searched which does not yield the position for the new z. Hence if there are m elements in the queue and n of these are searched for each z being added (including the element which reveals the location for the new z), we must add 25 X (n-1) T-periods for each z placed onto the queue. Additional time is needed as well, but the searching accrues most of the queueing time. Thus for each block:

$$T_{zqueadd} = (Z \text{ queued}) \text{ X } [72 + (25) \text{ X } (n-1)] \quad (5)$$

Hence if we combine equations (3), (4), and (5) we arrive at what seems to be a reasonable approximation equation for systems of more than 6000 gates:

$$T_{resident} = (68 + 35F)x + 29y \quad (3)$$

$$T_{pre} = (44) \text{ X } (4 + F) \text{ X (block size)} + 84 \text{ X (z prequeued)} + 94 \quad (4)$$

$$T_{zqueadd} = (\text{z queued}) \text{ X } [72 + (25) \text{ X } (n - 1)] \quad (5)$$

$$T_{overall} = (T_{resident} + T_{pre} + T_{zqueadd}) \text{ X (\# blocks loaded)}$$

(Where $T_{resident}$ is interpreted such that x and y are associated with block size instead of system size)

Substituting we get:

$$T = (\text{\# blocks in system}) \text{ X } \{(44) \text{ X } (4 + F) \text{ X (block size)} + 84 \text{ X (z prequeued)} + 94 + (\text{z queued}) \text{ X } [72 + (25) \text{ X } (n - 1)] + (68 + 35F)x + 29y\} \quad (6)$$

Equation (6) is the general form of our equation. Now for the following example, we will make some assumptions concerning the system under consideration. First we need some way to approximate, as closely as possible, the number of output gates (z's) which are prequeued for a block by other blocks, and as well, the number which are queued by each block. If we assume a square system (as discussed previously for the resident system configuration) then the number of outputs for a block = $\sqrt{\text{block size}}$. Additionally if 5% of those outputs are changing, then we can assume 5% of $\sqrt{\text{block size}}$ as a reasonable estimate for the number of z's to be prequeued for and/or queued by a given block.

Thus: z prequeued = z queued = $.05\sqrt{\text{block size}}$. (7)

Similarly we can say that the number of logic levels in the system

= y = $\sqrt{\text{block size}}$ (8)

(based upon a square system configuration). Now we must gain an understanding of n, the number of elements in the queue which are searched in order to add each z queued into the queue. We know from (7) above that for each block, 5% of $\sqrt{\text{block size}}$ is the number of gates queued by that block. Thus (# blocks in system) X ($.05\sqrt{\text{block size}}$) gives the maximum length of the z-processing queue at any time. Now we further assume that on the average, 50% of the queue needs to be searched for any given z.

Hence: n = ½(# blocks in the system) X ($.05\sqrt{\text{block size}}$) (9)

Combining equations (6), (7), (8), and (9) we get:

$$T = (\#\text{ blocks in System}) \times [(44) \times (4 + F) \times (\text{block size}) + (84) \times (.05\sqrt{\text{block size}}) + 94 + (.05\sqrt{\text{block size}}) \times \{72 + (25) \times [(\tfrac{1}{2}) \times (\#\text{ blocks in System}) \times (.05\sqrt{\text{block size}}) - 1]\} + (68 + 35F)x + 29\sqrt{\text{block size}}].$$

Combining terms gives:

$$T = (\#\text{ blocks in System}) \times \{94 + (44) \times (4 + F) \times (\text{block size}) + (.05\sqrt{\text{block size}}) \times [156 + (12.50) \times (\#\text{ blocks in system}) \times (.05\sqrt{\text{block size}}) - 25] + (68 + 35F)x + 29\sqrt{\text{block size}}\}$$

$$= (\#\text{ blocks in System}) \times [94 + (44) \times (4 + F) \times (\text{block size}) + 7.8\sqrt{\text{block size}} + (0.03125) \times (\#\text{ blocks in system}) \times (\text{block size}) - 1.25\sqrt{\text{block size}} + (68 + 35F)x + 29\sqrt{\text{block size}}].$$

So $T = (\#\text{ blocks in system}) \times \{94 + (\text{block size}) \times [(44) \times (4 + F) + (0.03125) \times (\#\text{ blocks in System})] + 35.55\sqrt{\text{block size}} + (68 + 35F)x\}$. (10)

Equation (10) is valid for all x and F in all systems meeting our initial assumptions. But in general we wish to use this as an aid in determining if this type of system is feasible. Hence let us further assume that the number of gates changing in the system at any given time is 5%. Thus x = 5% of the block size. Furthermore, assume a fan-out factor of two output gates per gate so F = # z's/x = 2.0. Substituting these into equation (10) we get:

$$T = (\#\text{ blocks in System}) \times \{94 + (\text{block size}) \times [(44) \times (4 + 2.0) + (0.03125) \times (\#\text{ blocks in system}) + (.05) \times (68 + 35 \times 2.0)] + 35.55\sqrt{\text{block size}}\}.$$

In simplifying this equation, for the purpose of understanding the nature of this relation and its applicability to real systems, we assume the most simple case in which block processing occurs sequentially without interblock feedback. This means that the total system size = (# blocks in the system) X (block size), and hence we can substitute system size into the equation for the term which contains this product. Realizing that this is not the general case, it is understood that for more complex systems, the time T will be greater than that which is given in this equation.

Thus by combining terms and substituting system size appropriately we get:

$$T = (\#\text{ blocks in System}) \times [94 + (0.03125) \times (\text{system size}) + (270.9) \times (\text{block size}) + (35.55\sqrt{\text{block size}})]. \quad (11)$$

Equation (11) gives T (in T-periods/cycle) for simple case systems with F = 2.0 and 5% of the system changing. An example of its use follows.

Example of Non-Resident System:

System size = 12000 gates.

Block size = 6000 gates.

(hence # blocks = 2)

$$T = 2[94 + (0.03125)(12000) + (270.9)(6000) + (35.55)(77.46)]$$

$$= 2(94 + 375 + 1,625400 + 2753.7)$$

$$= 2(1,628,622.7) = 3,257,245.4 \text{ T-periods}$$

$$= 260,579,630\text{ns}$$

Or a slow down factor of 260,580:1 for a 1μs cycle machine.

(2,605,796:1 for a .1μs machine)

Compare this to the 3491.7:1 slow down for a resident system of 6000 gates. (34,917:1 for .1μs system)

APPENDIX C

NANOCODE FOR BEST CASE TIMING ESTIMATE

This appendix contains the "minimum" nanocode to implement the processing necessary for the algorithm as defined in section III. The structure shown in Figure III-3 is assumed and the following Local Store (LS) register assignments are also assumed:

LS register	contents
x	address of current x gate info word
y	address of address of current z gate info word
z	address of current z gate info word
w	MLINK
a	constant integer 2
b	scratch

The x-gate processing (read of info word) and multi-way branch assumes: gate info word address in LS register x.

XPROC:	....	BRANCH(N. + 1) KA = x KB = 31.	
	X...	KA → FCIA, KB → FCOD	Set up to read info word into R31.
4T	.X..		CS bus wait.
	..S.	READ CS(CIA), GATE CS, READ NS, GATE NS	R31← gate info word.
:	....		
	X...	B→ FIDX	Set up top 3 bits of NS address.
	.X..	INCF→ FIDX	A 1 in bit Ø signifies x processing.
5T	..X.	LOAD NPC(CS)	Set up NPC for branch based on FIDX and top 7 bits of gate info word.
	...S	READ NS, GATE NS	Branch through micro-op-code.

9T

Read of CLINK for this x assumes:

- gate info word address is in LS register x
- constant 2 is in LS register a
- scratch register is LS register b
- gate info word address for next output (z gate) is in LS register z

```
:    ....  BRANCH(ZPROC)
           KALC = ADD
           KA = x
           KB = a
           KX = b
           KT = y

     X...  KB → FAIR, KA → FAIL,        Set up to get b = x+a  (z = x+2)
           KX → FAOD

     .S..  GATE ALU, KT → FCOD,         Register b ← x+2 (addr CLINK into
           KX → FCIA.                   b).  Set up to read CLINK value
                                        into y.

6T   ..X.                               CS bus wait.

     ...X  READ CS(CIA), GATE CS,       y ← CLINK (address output gate
           READ NS, GATE NS             list).
```

6T This then proceeds to Z processing (ZPROC).

Set up for next x-gate assumes:

- current gate info word address is in LS register x
- scratch register is LS register b

```
:    ....   BRANCH(XPROC)
            KALC = INCR LEFT
            KA = x
            KB = b

     X....  KA → FAIL,  KB → FAOD,     Set up to increment address to
            SET CIH                    get addr of LINK.

6T   .S..   GATE ALU, KB → FCIA,       Scratch register b ← x+1.  Now
            KA → FCOD                  set up to read that LINK.

     ..X.                              CS bus wait.

     ...S   READ CS(CIA), GATE CS,     x ← address next gate info word
                                        (LINK of current word).

            READ NS, GATE NS           Then go to XPROC.

-----
6T
```

z-gate processing set up and branch assumes:

- address of the address of this z gate info word is in LS register y
- address of this z gate info word in LS register z

ZPROC:	....	BRANCH(N. + 1) KALC = INCR LEFT KA = y KB = z KX = 31.	
	X...	KB → FCOD, KA → FCIA, KA → FAIL, SET CIH	Set up to read gate info word for this gate; and to increment the address to point to next gate address.
5T	.X..	KA → FAOD	Want to write the new address back.
	..S.	READ CS(CIA), GATE CS, GATE ALU, READ NS	Register z ← addr gate info word. y ← y+1 (Next gate addr).
	...X	KB → FCIA, KX → FCOD, GATE NS	Set up to read gate info word into R31.
:	....	BRANCH(N.+1)	
	X...		CS bus wait.
4T	.S..	READ CS(CIA), GATE CS, READ NS	R31 ← gate info word for this z.
	..X.	B → FIDX, GATE NS	Set up top 3 bits of NS address.
:	....		
3T	X...	LOAD NPC(CS)	Set up multi way z branch.
	.S..	READ NS, GATE NS	And go.
12T			

Addition of z-gate to link queue assumes:

- This z gate info word address is in LS register z
- Scratch register LS is register b
- MLINK is in LS register w

```
:   ....   BRANCH(N.+1)
           KALC = INCR LEFT
           KA = z
           KB = b
           KX = w
           KS = PASS LEFT
       X...  KA → FAIL, KB → FAOD,        Set up to find addr of LINK
             SET CIH                      for this z.
       .S..  GATE ALU, KB → FCIA,         Scratch register b ← addr LINK.
             KA → FCID,                   Now set to write MLINK into this
                                          LINK,
6T           KA → FAIL                    And to set MLINK to address of
                                          this z info word.
     ..X.    G(G KS); G → KALC            Change ALU to PASS LEFT this gate
             KX → FAOD                    info word addr to MLINK.
     ...S    WRITE CS(CIA),               Set LINK to MLINK.  Set MLINK to
             GATE ALU,                    addr of this gate info word;
             READ NS, GATE NS             And continue.
```

6T

For each x-gate, the last z output test assumes:

- LS register contains address of this gate info word
- bit 17 is set for the last z in list

	:	BRANCH(XPROC)	
		ALT BRANCH	
		KALC = PASS LEFT	
		KT = SIGN	
		KA = z	
	X...	KA → FAIL	Test sign bit.
4/6T	.X..	LOAD NPC(SEQ)	ALT. BRANCH to next word.
	..S.	READ NS, GATE NS(T)	Branch to XPROC for sign = 0.
	...S	READ NS, GATE NS	Otherwise continue to N.+1.
	:	BRANCH(ZPROC)	
2T	S...	READ NS, GATE NS	
4/8T			

NASA National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-181641		

4. Title and Subtitle	5. Report Date
Digital Avionics Design and Reliability Analyzer	February 1981
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
	10. Work Unit No. 505-66-21-03

9. Performing Organization Name and Address	11. Contract or Grant No.
Martin Marietta Denver, CO 80201	NAS1-15780
	13. Type of Report and Period Covered

12. Sponsoring Agency Name and Address	13. Type of Report and Period Covered / 14. Sponsoring Agency Code
National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225	Contractor Report

15. Supplementary Notes

Technical Monitor: Gerard E. Migneault
Langley Research Center

16. Abstract

This document contains the description and specifications for a digital avionics design and reliability analyzer. Its basic function is to provide for the simulation and emulation of the various fault-tolerant digital avionic computer designs that are developed. It has been established that hardware emulation at the gate-level will be utilized. The primary benefit of emulation to reliability analysis is the fact that it provides the capability to model a system at a very detailed level. Emulation allows the direct insertion of faults into the system, rather than waiting for actual hardware failures to occur. This allows for controlled and accelerated testing of system reaction to hardware failures. There is a trade study which leads to the decision to specify a two-machine system, including an emulation computer connected to a general purpose computer. There is also an evaluation of potential computers to serve as the emulation computer.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Reliability Analysis Digital Emulation	Unclassified-Unlimited Subject Category 62

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	153	